



## UvA-DARE (Digital Academic Repository)

### Specification-centric multi-agent systems

Esterhuysen, C.A.

**Publication date**

2025

**Document Version**

Final published version

[Link to publication](#)

**Citation for published version (APA):**

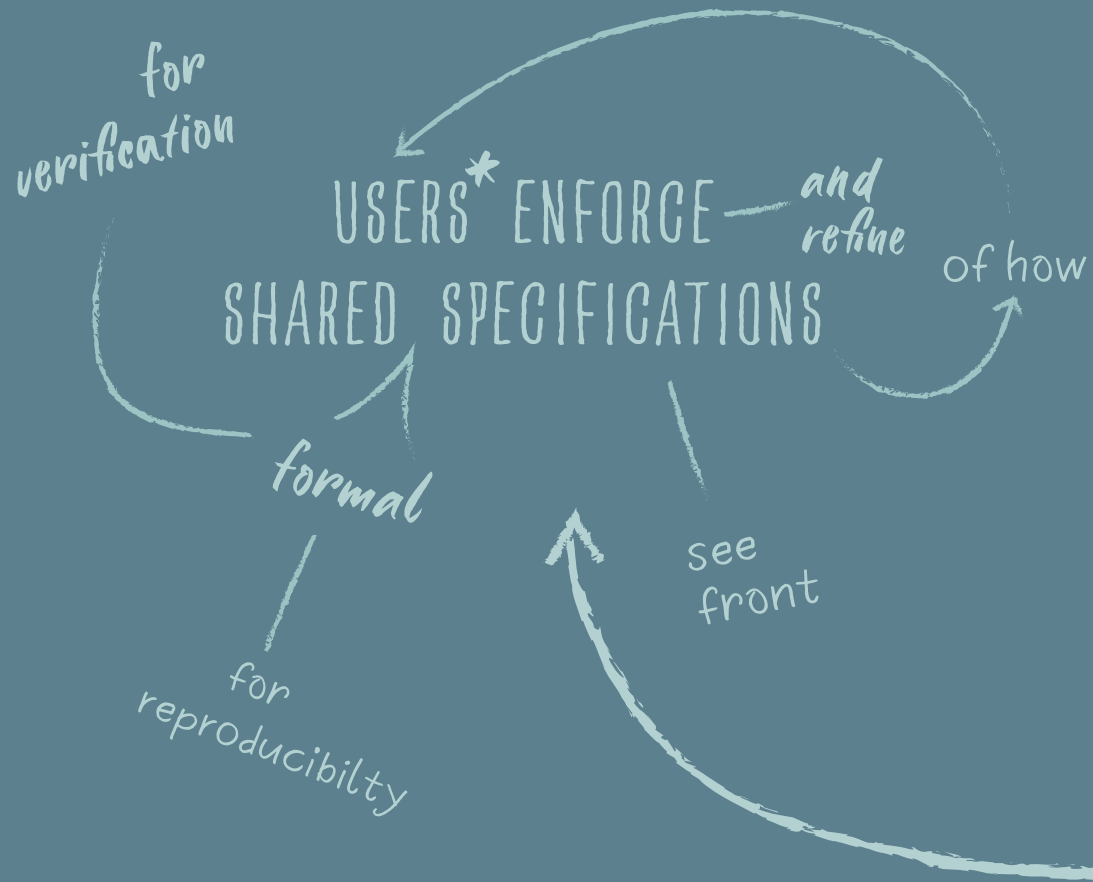
Esterhuysen, C. A. (2025). *Specification-centric multi-agent systems*. [Thesis, fully internal, Universiteit van Amsterdam].

**General rights**

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

**Disclaimer/Complaints regulations**

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.



**APPROVED BY**  
 Amy Bob DAN

\* and automated services

Specification-Centric Multi-Agent Systems

Christopher Arno Esterhuyse

# SPECIFICATION -CENTRIC MULTI-AGENT SYSTEMS

- laws
- policies
- PROTOCOLS

- DISTRIBUTED
- autonomous
- federated

# Specification-Centric Multi-Agent Systems

## ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor  
aan de Universiteit van Amsterdam  
op gezag van de Rector Magnificus  
prof. dr. ir. P.P.C.C. Verbeek

ten overstaan van een door het College voor Promoties ingestelde commissie,  
in het openbaar te verdedigen in de Agnietenkapel  
op maandag 8 september 2025, te 13.00 uur

door Christopher Arno Esterhuysen  
geboren te Richardsbaai

***Promotiecommissie***

<i>Promotor:</i>	prof. dr. ir. C.T.A.M. de Laat	Universiteit van Amsterdam
<i>Copromotor:</i>	dr. L.T. van Binsbergen	Universiteit van Amsterdam
<i>Overige leden:</i>	prof. dr. D. Amyot	University of Ottawa
	prof. dr. B. Combemale	University of Rennes
	prof. dr. S.J.L. Smets	Universiteit van Amsterdam
	prof. dr. S. Klous	Universiteit van Amsterdam
	dr. V.O. Degeler	Universiteit van Amsterdam
	dr. A.M. Oprescu	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

---

## Acknowledgements

**Professional Acknowledgements** Work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and was funded by the projects AMdEX-Fieldlab (Kansen Voor West EFRO grant KVV00309) and AMdEX-DMI (Dutch Metropolitan Innovations ecosystem for smart and sustainable cities, made possible by the Nationaal Groeifonds).

**Personal Acknowledgements** First of all, thanks to my PhD co-supervisors; Thomas, I appreciate your trust in me, your genuine joy in most research activities, and your double standard of taking endless work on yourself, but not expecting it of others; Cees, I appreciate your open-mindedness and patience with me while I learn(ed) the ropes, and your no-nonsense approach to all things except for matters of humour. Thanks to you, and to my thesis opposition committee, for accepting the responsibility; to you all, I dedicate the following:

*If I had more time, I would have written you a shorter [thesis]. – Blaise Pascal, allegedly*

Thank you to my teachers at the VU, for being generous with your expertise and time: Natalia, Wan, Jörg, Femke, Clemens, Gunnar, Henri, Andy, Alexandru, Sanjay, and Herbert.

Thanks to my old colleagues at CWI, with whom I had many formative interactions over white-beers and -boards. Firstly, thanks to Gunnar, Jasmijn, Alex, Roy, Frank, and Farhad for letting my foot in the door at CWI, twice (*i.e.*, both feet). Frank, you did your best to stop us ‘hanging ourselves with our milestones’. Farhad, I admire your rhetoric and your big-picture perspective. Sung-Shik, your easy demeanour turns problems into steps and conversation into inspiration; *e.g.*, our brief discussion at PLNL’23 made me rethink ‘explainability’. Thanks to the Reowolf gang, for the camaraderie that outlived our shared projects and contracts. Bonus thanks to Luc, Mathé, Kasper, and Jinting. Hans, I admire your initiative, boldness, and relentless pursuit of knowledge. And Benjamin, I admire your clarity in thought except where it is hopelessly obscured by artful farce.

Thank you to my colleagues at the UvA, who were curious and friendly without exception. First, thanks to my (unofficial) EPI fellows for getting me started: Milen, Tim, Jamila, Onno, Lourens, Adam, and Paola. Lourens, I learned a lot from studying your Mahiru system. Second, thanks to my office mates, which whom I shared the bond that can only be forged in mutual distraction: Cyril, Na, Marco, Daphnee, Floris-Jan, Yuri, Marten, Shashank, and Adnan. Thanks to Giovanni, for sustaining our group cohesion despite leaving CCI, and

for serendipitously suggesting that I look into Clingo. Third, thanks to all of my other colleagues whose insight and companionship I sought out: Damian, Merrick, Nina, Ana, Tom, Andrea, Sander, Sander, Kostas, Divya, Tomasz, Viktoriya, Clemens, Pooya, Angelos, Jose, Paul, Tommaso, Ziyang, Gabriel, Joseph, Koen, Imre, Misha, Leonardo, Yixian, Georgios, Kyrian, Linus, Romke, Jelle, Lukas, Marius, Sudaksh, Florine, Clemens, Julius, and Mike. Special thanks to Tim and Milen, for always being there when I look behind me. Thanks to the AMdEX gang for the easy work environment, and to my fellows in CCI, for the constant source of community and goofiness, as my collaborators on research papers, and my companions in travels to conferences. Thanks to my paranymphs, Henry and Tim, for being my support – my rock, and stone-faced defenders, in my final rite of passage. A redundant thanks to Tim, for finding himself at the intersection of these groups; many of us owe you much for your self-, thank-, and tire-less work. And thanks to Damian, for befitting your role as my academic brother; thanks for not making me wait in translating my Summary.

Thanks to friends beyond UvA and CWI for their companionship in the gaps between sleep and work, for each playing parts in my life, in your own way. Thanks to Jaco, Anthony, Cass, Lloyd, and Henry for being my oldest friends, whose qualities haven't changed after all this time. Thanks to Michael & Uwe for getting me started in Europe. Thanks to Angel, Henry, Ronja, Vivi, and Violet, for seeing me through uncertain times. Thanks to Dalia, for being the kind of attentive and thoughtful friend I can only aspire to be. Miscellaneous thanks to James, Alex, Becky, Peter, Berend, Jasmine, Axel, Vero, Melina, Julie, Maike, Yashna, Paula, Melanie, Yan, and everyone I left out by mistake. Thanks again to Henry, Roy, Tim, and Ronja, as my closest companions (demonstrably!) at the time of writing.

Thanks to all my family, whose affection and influence made me who I am today. Since I've kept her waiting long enough, I first thank my *Oma*; *na siehst du? Ich hab's endlich geschafft!* To elders Lothar, Heinz, Andrea, Wilhelm, Brigitte, Dian, Abrie, Bernie, Adrienne, Jen, Rich, Gordon, Les, and Adele, thanks for doing for me what parents do. For example, this is just another way an onlooker might confuse Brigitte with my mom. Thank you to Sabine, Jan, Ronja, Paul, and Jaspar, for being an excellent surrogate family for more than a decade. Thanks to my most literal parents: Inge, Grant, Jonathan ('B.C.'), and Sharon; despite all the other support, I needed yours all the same; I appreciate that you made me feel free in going my own way, but at the same time, by sharing in my achievements, you made them all the more worthwhile. Thank you to all my (extended) siblings and cousins, who did all sorts of things, sometimes helpful and sometimes entertaining, which is just what I needed: Chelsea, Suzanne, Marek, Thomas, Geli, Charl, Chantal, Tatjana, Martin, Taygan, Keiryn, Caitlin, Nathan, Ciara, Jethro, Leo, Arya, (younger) Thomas, Chloe, Ashlyn, Amely, Kyra, Brook, Luke, and Dane. And thanks to the partners of the above, who are family just the same: Nate, Mila, Jordan, Willem, Elise, (older) Lloyd, Tiffany, and Vika.

Finally, the most special thanks to Ronja for designing the poster (Figure 1.4) and illustrations (Pages 280–281); and for giving me feedback for all my academic efforts so far; and for the years of loving support and companionship before and besides (that's good!) and then going along with this idea of carrying on (that's better!); for all of that, I can only dedicate this, my (final) thesis to you. Hat trick!

---

# Contents

<b>Acknowledgements</b>	<b>iii</b>
<b>Contents (you are here)</b>	<b>v</b>
<b>1 General Introduction</b>	<b>1</b>
1.1 The Goal: Flexible and Accountable Data Exchange . . . . .	1
1.2 The Means: Piecewise Specification and Enforcement . . . . .	3
1.3 Structure and Contributions of the Thesis . . . . .	5
1.4 Narrative Thread through the Chapters . . . . .	7
<b>2 eFLINT Semantics and Model-Checking via Clingo</b>	<b>17</b>
2.1 Introduction . . . . .	18
2.2 Translation Source Language: eFLINT . . . . .	22
2.3 Translation Target Language: Clingo . . . . .	24
2.3.1 Clingo in Context: Logic- and Answer-Set Programming . . . . .	24
2.3.2 Features of The Target Fragment of Clingo . . . . .	24
2.4 State Traces . . . . .	27
2.4.1 A Primer on the Event Calculus . . . . .	27
2.4.2 State Traces, Embedded in Clingo . . . . .	27
2.4.3 State-Internal Rules atop State Traces . . . . .	29
2.5 Core eFLINT . . . . .	29
2.6 Specification Translation to Core eFLINT . . . . .	32
2.6.1 Abstract Syntax . . . . .	32
2.6.2 Instance Types and Static Semantics . . . . .	34
2.6.3 Restricted Normal Form . . . . .	35
2.6.4 Dynamic Semantics of eFLINT Specifications . . . . .	36
2.7 Scenario Translation to Core eFLINT . . . . .	39
2.7.1 Use Case: Check if a Scenario is Compliant . . . . .	39

2.7.2	Use Case: Search for Satisfactory Scenarios . . . . .	39
2.8	Implementation Overview . . . . .	40
2.9	Implementation Evaluation . . . . .	42
2.9.1	Correctness Evaluation . . . . .	43
2.9.2	Performance Evaluation . . . . .	46
2.10	Related & Future Work . . . . .	52
2.11	Conclusion . . . . .	55
<b>3</b>	<b>Seaso: A Language for the Controlled Multi-Agent Specification of Data Exchange Systems</b>	<b>57</b>
3.1	Introduction . . . . .	58
3.2	Background . . . . .	61
3.2.1	Data Exchange Systems . . . . .	61
3.2.2	Logic Programming . . . . .	62
3.2.3	Normative System Specifications . . . . .	63
3.3	Definition of the Seaso Language . . . . .	65
3.3.1	Informal Language Overview . . . . .	65
3.3.2	Meta-Language Notation and Abstract Seaso Syntax . . . . .	67
3.3.3	Static Semantics: Program Composition and Well-Formedness	68
3.3.4	Dynamic Semantics: Inference and Denotation . . . . .	70
3.3.5	Concrete Syntax . . . . .	73
3.3.6	Finite and Unique Denotations in Finite Steps . . . . .	74
3.4	Case Study: Specifying Data Exchange . . . . .	77
3.4.1	Agents Transfer Data . . . . .	77
3.4.2	Data Computation . . . . .	78
3.4.3	Roles of Data in Computation: Workflows and Plans . . . . .	79
3.4.4	Roles of Agents in Computation: Data Exchange Archetypes .	79
3.4.5	Data Exchange Norms . . . . .	82
3.4.6	Dynamic (Knowledge of) Concrete Systems . . . . .	82
3.5	Discussion . . . . .	84
3.5.1	Reasoning about Rules . . . . .	84
3.5.2	Reasoning about Atom Values . . . . .	85
3.5.3	Reasoning about Time and Dynamic Systems . . . . .	86
3.5.4	Reasoning about Domains and Signatures . . . . .	87
3.5.5	Composition Control with Seal Statements . . . . .	88
3.5.6	Finite Products with Infinite Extensions . . . . .	88
3.5.7	Interpretation of Atom Valuations . . . . .	89
3.5.8	Requirements Revisited . . . . .	91

3.5.9	Prototype Implementation . . . . .	92
3.6	Related work . . . . .	92
3.6.1	Logic Programming Languages . . . . .	92
3.6.2	Approaches to Consistency-Preserving Program Composition . . . . .	93
3.6.3	Executable Norm Specification Languages . . . . .	93
3.6.4	Software Modelling Languages . . . . .	95
3.6.5	Goal-Oriented Requirements Engineering . . . . .	95
3.7	Conclusion . . . . .	96
<b>4</b>	<b>Cooperative Specification via Composition Control</b>	<b>99</b>
4.1	Introduction . . . . .	100
4.2	Definitions up to Composition Control . . . . .	102
4.2.1	Language Schema . . . . .	103
4.2.2	Cooperative Specification . . . . .	104
4.2.3	Composition Control . . . . .	105
4.3	Composition Control Features in Datalog . . . . .	105
4.3.1	The Datalog Language . . . . .	105
4.3.2	Composition Control in Datalog Variants . . . . .	107
4.4	Adapting Existing Languages for Cooperative Specification . . . . .	111
4.4.1	Running Example Usage Scenario . . . . .	111
4.4.2	Cooperative Specification in Alloy . . . . .	111
4.4.3	Cooperative Specification in eFLINT . . . . .	116
4.5	Designing Bespoke Cooperative Specification Languages . . . . .	125
4.5.1	Desirable Qualities in Cooperative Specification Languages . . . . .	125
4.5.2	Seaso: Static and Robust Cooperative Specification . . . . .	130
4.5.3	Slick: Dynamic and Flexible Cooperative Specification . . . . .	133
4.6	Related Work . . . . .	135
4.6.1	Composition Control in General-Purpose Languages . . . . .	135
4.6.2	Smart Contract Languages . . . . .	136
4.6.3	Powerful Formalisms and Formal Verification . . . . .	137
4.6.4	Abstract Argumentation Frameworks . . . . .	137
4.7	Discussion . . . . .	138
4.8	Ongoing & Future Work . . . . .	139
4.9	Conclusion . . . . .	140
<b>5</b>	<b>Formal Foundations for Reowolf</b>	<b>143</b>
5.1	Introduction . . . . .	143
5.2	User Communication Sessions via Connectors and PDL . . . . .	146
5.3	Protocol Description Language (PDL) . . . . .	148

5.3.1	PDL Syntax and Small-Step Semantics . . . . .	148
5.3.2	Composite Protocols and (A)synchrony . . . . .	150
5.3.3	Linear Execution Traces and Accepted Behaviour . . . . .	151
5.3.4	Behaviour Constructed from PDL Protocols . . . . .	155
5.4	PDL Runtimes . . . . .	158
5.4.1	A Specification of PDL Runtimes . . . . .	158
5.4.2	Example: The Silent PDL Runtime . . . . .	159
5.5	Future Work . . . . .	160
5.5.1	Formalising the Rest of the Connector Runtime . . . . .	160
5.5.2	Exploring Variations of the PDL . . . . .	161
5.5.3	Definition, Analysis, and Optimisation of PDL Protocols . . . . .	162
5.6	Related Work . . . . .	163
5.6.1	Related to Behavioural Specification with the Reowolf PDL . . . . .	163
5.6.2	Related to Network Programming with Reowolf Connectors . . . . .	164
5.7	Conclusion . . . . .	166

## 6 JustAct: A Framework for Policy-Regulated Multi-Domain Data

	<b>Processing</b>	<b>167</b>
6.1	Introduction . . . . .	168
6.2	Background . . . . .	170
6.2.1	Distributed Systems and Algorithms . . . . .	170
6.2.2	Multi-Agent Systems and Autonomy . . . . .	170
6.2.3	Policy Specification Languages . . . . .	171
6.2.4	Logic Programming and Datalog with Weak Negation . . . . .	172
6.3	The JustAct Framework for Multi-Agent Runtime Systems . . . . .	173
6.3.1	Statics . . . . .	174
6.3.2	Dynamics . . . . .	176
6.4	Implementation: Generic Data Exchange Runtime System . . . . .	180
6.4.1	Design Considerations . . . . .	180
6.4.2	The Data Plane . . . . .	182
6.4.3	The Control Plane . . . . .	182
6.5	The Slick Policy Language & Interpreter . . . . .	188
6.5.1	Design Considerations . . . . .	188
6.5.2	The Slick Language . . . . .	190
6.5.3	Connecting Policies to the Runtime System . . . . .	192
6.5.4	The Slick Interpreter Implementation . . . . .	194
6.6	Case Study: Processing Distributed Medical Data . . . . .	195
6.6.1	The Case: The Brane Component of the EPI Framework . . . . .	195

6.6.2	Initialising our Runtime System for Brane Scenarios . . . . .	197
6.6.3	Usage Scenarios . . . . .	202
6.6.4	Evaluation of the Case Study . . . . .	211
6.7	Discussion of the Framework . . . . .	215
6.7.1	Strengths of the Framework . . . . .	215
6.7.2	Limitations of the Framework . . . . .	216
6.8	Related work . . . . .	218
6.8.1	Smart (Policy) Ledgers atop Blockchains . . . . .	219
6.8.2	Curie . . . . .	219
6.8.3	Trust Management . . . . .	220
6.9	Conclusion . . . . .	221
<b>7</b>	<b>General Conclusion</b>	<b>223</b>
7.1	Reflection on the Thesis Contributions . . . . .	223
7.2	Avenues for Future Work . . . . .	225
	<b>Summary</b>	<b>227</b>
	<b>Samenvatting</b>	<b>228</b>
	<b>Glossary and Abbreviations</b>	<b>229</b>
	<b>Appendices</b>	<b>239</b>
A	An Account of the Situation, Event, and Fluent Calculi . . . . .	239
B	Example 2.1 Translated from eFLINT to Clingo in Full . . . . .	240
C	One Seaso Denotation Computed in Full . . . . .	244
D	Formalising Extra Data Exchange Archetypes in Seaso . . . . .	245
E	Reowolf Terms in Chapter 5 vs. the Formalism in Coq . . . . .	246
F	Gentle Introduction to (Our Reowolf Formalism in) Coq . . . . .	248
	<b>Bibliography</b>	<b>254</b>
	<b>Titles in the IPA Dissertation Series since 2022</b>	<b>276</b>



# General Introduction

## 1.1 The Goal: Flexible and Accountable Data Exchange

In recent years, a multitude of academic and industrial projects have developed consortia, infrastructures, and software systems for various cases of *data exchange*: the controlled sharing and processing of valuable data across organisational boundaries. Consider the following examples of data exchange platforms and projects.

1. *AutoMat* is a marketplace for sharing automotive vehicle sensor data across vehicle brands and models, to improve autonomous vehicle systems, *e.g.*, to keep traffic routing informed by (changes in) local weather and road quality [PWZ<sup>+</sup>17].
2. *The ICARUS framework* specialises in brokering aviation data exchange agreements, by handling the sharing policies and intellectual property rights [BPP<sup>+</sup>19].
3. *The Health Research and Innovations Cloud* is under development, aiming to facilitate safe and ethical sharing of health research data across the EU [AAA<sup>+</sup>20].
4. *The International DIPG Registry* is used to share anonymised medical data, specifically on the DIPG disease, toward the development of new treatments [BBL<sup>+</sup>17].
5. Developed by the OpenAIRE project [SK12], *Zenodo* is a platform for unrestricted sharing of research papers and other digital artefacts<sup>1</sup>, *e.g.*, AI models, source code, experimental data, and project documentation [GN25].

The mass of these projects reflects the promise of their efforts; there is a great deal of value currently locked away in large silos of private data. The variety of these projects reflects the variety of (application) domains with potential for data sharing, *e.g.*, sharing the scarce DIPG data is necessary to develop treatments, and vast amounts of aircraft sensor data is needed to train accurate AI models for predictive aircraft maintenance [LdPMS25, LTK<sup>+</sup>18]. All these projects are complicated by the

---

<sup>1</sup>In fact, this thesis makes extensive use of Zenodo for disseminating the source code, documentation, and experimental data behind the main contributions. Please see Figure 1.2 for the details.

crucial need to carefully regulate the exchange and usage of the data, to safeguard the privacy of medical patients and industrial secrets, as the case may be. For example, [BBL<sup>+</sup>17] presents their process of reviewing user applications, and their procedure for anonymising shared DIPG data, toward the satisfaction of the HIPAA (an act of the United States Congress regulating access to medical patient data).

Several projects have made some progress in developing and validating software systems that meet all the stakeholders' requirements. But progress is impeded by the difficulty in eliciting these requirements. Different stakeholders are concerned with different facets of the system, and formulate their requirements in different terms [GRJ22]. For example, in the automotive industry, different stakeholders must cooperate, despite being specialised in the legal regulations of different countries [LTK<sup>+</sup>18]. Often, legal experts and system administrators must reconcile their notions of *privacy*, and how they pertain to the sharing of data. The elicitation process also reveals conflicts between requirements, causing delays while they are renegotiated [BZ19]. Indeed, *requirements negotiation* is an entire area of study [BBHL94] offering several solutions [RM22, KKS21, BZ19, Suh19]. The most difficult cases arise where conflict is intrinsic, and the stakeholders must compromise. For example, scientists are incentivised to access as much sensitive medical data as possible, to enable significantly meaningful results; *e.g.*, this necessity motivates the sharing of DIPG data [BBL<sup>+</sup>17]. Meanwhile, medical patients are incentivised to keep their sensitive data private, because even the most carefully controlled sharing of their data incurs some risk that it is misused in the future [GKV<sup>+</sup>20]. For example, medical patients with serious conditions harm themselves by revealing the details of their illnesses to their (future) medical insurers [HR00, AS15]. Finally, the requirement elicitation and reconciliation effort never ends, because stakeholder requirements change over time. For example, data processor requirements change as particular vendors come and go over the lifetime of the system, and legal regulations change to update requirements on the software, for example, in its guarantees about user privacy [TCM24, Shi11].

In abstract, two notable requirements never seem to change. Firstly, data exchange systems must remain flexible to the user requirements being enforced, such that the system can operate, despite ever-changing users and user requirements. Secondly, these systems must systematically enforce the accountability of users to their actions, to mitigate the high risk of misuse of valuable data in an ever-changing environment.

The EU commission has mandated the coordination and inter-operation of data exchange projects, for the sake of streamlining, inter-connecting, and de-duplicating their work, while improving the quality and longevity of their results [Eur20]. New projects and consortia have responded with large-scale consultation and standardisation efforts, aiming to expose and create relations between existing projects and solutions.

The *Amsterdam Data Exchange (AMdEX)* is such a project (see <https://amdex.eu>), emphasising solutions that generalise many use cases and application domains.

The grants for AMdEX-Fieldlab and AMdEX-DMI funded my research. So the question was: what theory and tooling aligns with the EU mandate of generic solutions to the problem of flexible and accountable data exchange?

## 1.2 The Means: Piecewise Specification and Enforcement

This thesis overviews my contributions to the greater research effort that is ongoing in my research group of the University of Amsterdam (UvA<sup>2</sup>), and which reflects my background and the influence of my peers.

This thesis takes a consistent approach to challenges of facilitating, reasoning about, and controlling the operation of complex, distributed, multi-agent, (data exchange) software systems. Many problems are framed as the need for agents to effectively *specify* their requirements on the behaviour of the system. For example, their selected specification language must be able to faithfully capture their requirements.

**What is a Specification Language?** Ultimately, every specification in every language has the same main purpose: it is a concrete and communicable artefact, whose value lies in its formalised meaning in relation to *behaviour*, *e.g.*, of a data exchange system [SMHB98, MK20]. Ultimately, the meaning of any specification is in its discrimination between the behaviours it *accepts* from those it does not.

How we use these specifications varies from case to case. The simplest case is perhaps the selection of a specification, because it assigns a *value* to each system behaviour. Depending on the context, these values express requirements, expectations, observations, or predictions of (future) behaviour. Thus we can frame the problem of eliciting stakeholder requirements on the data exchange system as their selection of a specification. For example, this is the main concern of Chapter 3, where stakeholders take turns formalising each of their requirements in a shared specification.

Once a specification is available, there are many ways it can be put to good use. Often, we *enforce* specifications in a real system: we *prevent* or *correct* system behaviour which is specified to be unacceptable. For example, in Chapter 6, agents specify which actions they permit, and then auditors punish agents for acting without permission. In other situations, acceptance is systematically preserved. For example, in Chapter 5, agents specify which communication behaviour they accept, and then some behaviour is selected and realised, such that all agents are satisfied.

To make sense of these various usages and application contexts, this thesis borrows terminology from the literature. The following overviews the main terms.

---

<sup>2</sup>Please refer to the Glossary to clarify such names and abbreviations used in this thesis.

1. A *model* is an abstraction of a concrete (data exchange) system. Generally, our specification languages are also modelling languages, because they model the current state and acceptable system behaviours of the system. The precise roles of the model(s) in each specification depend on the language, so they vary between the chapters. For example, in Chapter 2, each specification models the system as it is, including which actions may happen next, and which ways it currently violates norms. And in Chapter 5, each specification models the current system state and models each communication pattern that would be acceptable next.
2. A *program* is some syntactic object, assembled from parts. A program is *executable*, whereupon it produces some result. In many chapters, specifications are the (syntactic) programs that users manipulate, and their execution results are (semantic) models that users reason about. For example, in Chapter 3, programs specify granular relationships between entities, and their execution applies logical reasoning to fill in any implied relationships. When programs are *deterministic*, the same programs necessarily always (re)produce the same results. In Chapter 6, agents rely on the determinism of their language, so that their explicit agreement on specifications implies their agreement on models. But in Chapter 5, programs are *nondeterministic*; users reason about the *set* of possible execution results, but execution ultimately selects one result in particular.
3. We use *norm* and *policy*<sup>3</sup> to refer to specifications of *normative* concepts from the social and legal literature and traditions. For example, in Chapter 2, specifications are policies because they capture obligations between institutional entities.
4. We use *protocol* as an alias for specification in Chapter 5, where behaviour is communication, *i.e.*, the (coordinated) passing of messages over time.

**How do we Define and Use Specifications?** We only consider specification languages which are defined in some well-understood mathematical formalism like a *set theory* or a *type theory*. As per tradition, we usually break each language into its *syntax* and its *semantics*. The former characterises how specifications are represented, built from smaller parts, and broken down again. The latter relates each specification to the semantic domain that we reason about, usually a model of data exchange systems. Many times throughout this thesis, we will define a new specification language as a syntax-semantics definition pair, often in terms of existing languages. This formal approach has many advantages. Firstly, by basing different languages on the same mathematical foundations, we have a starting point for their

---

<sup>3</sup>The term ‘policy’ is particularly overloaded in the literature. Often, it takes on an imperative connotation, where policies operationalise specifications of another kind. For example, this is the case in XACML policies, whose granular rules express enforceable conditions on access to resources.

comparison. Secondly, assuming we all agree on the mathematical foundations, they standardises the interpretation of specifications. For example, in Chapter 4, it suffices for the agents to agree on which specification is selected, because then they will certainly all reach the same conclusions when reasoning about its meaning.

Lastly, rigour is necessary to *automate* all of the aforementioned usages of specifications. We run into automation from the start, during the development process. We *implement* our language definitions in software tools, to aid (the reproducibility of) our experimentation. For example, all of our contribution chapters are backed by software artefacts: implementations of language compilers or interpreters, and experimentation scripts and results. But automation is also needed in application; human interaction with specifications and behaviour is mediated by automation at every level. For example, in Chapter 5, humans rely on a distributed software system to communicate their specifications over the network, and to realise communications. In Chapter 6, autonomous software agents act on the behalf of humans, creating new specifications, communicating them with their peers, and reasoning about them at runtime. This degree of automation is unavoidable in practice, where specifications become too large – and reasoning becomes too complicated – for humans to manage without machine assistance [End17]. In this context, humans rely on the rigour and correctness of the theory and its implementation for the systems to behave as intended.

**What is the Novelty?** This thesis consistently reflects a unique emphasis on specifications that can be extensively modified or composed, sometimes at the cost of other features. In every case, we use this to keep specifications aligned with changing requirements and expectations on the system. For example, Chapter 3 defines a language in which agents can refine the (semantic) model by composing new parts on the persistent and shared (syntactic) program. The later sections keep this incremental aspect, but connect more and more functionality to specifications.

### 1.3 Structure and Contributions of the Thesis

The main contributions of the thesis are spread over Chapters 2 to 6. Figures 1.1 and 1.2 lay out the research articles and supplementary artefacts (source code, proofs, and experimental results) which are the bases of these chapters. The contributions are laid out roughly in order that the underlying research efforts were *started*, because they significantly overlapped, and some even continue at the time of writing. Section 1.4, to follow, lays out the narrative thread through these chapters, but we summarise it here: all chapters concern the incremental specification of multi-agent software systems, but successive chapters introduce complications as we approach our goal.

Article Title	Citation	Venue	Status & Authorship	Thesis Relevance	
• A Stable Model Semantics for eFLINT Norm Specifications and Model Checking Scenarios	[EMvB25b]	GPCE'25	Published	1st	basis of Chapter 2
• Cooperative Specification via Composition Control	[EvB24]	SLE'24	Published	1st	basis of Chapter 4
• Reowolf: Synchronous Multi-Party Communication over the Internet	[EH19]	FACS'19	Published	1st	position of Chapter 5
• Formal Foundations for Reowolf: Multi-Party Sessions via Synchronous Protocol Programming	[ELHA25]	COORDI-NATION'25	Published	1st	basis of Chapter 5
• Exploring the Enforcement of Private, Dynamic Policies on Medical Workflow Execution	[EMvBB22]	ReWORDS, eScience'22	Published	1st	part of Section 6.6
• The EPI framework: A Data Privacy by Design Framework to Support Healthcare Use Cases	[KME <sup>+</sup> 24]	FGCS'24	Published	3rd	part of Section 6.6.1
• JustAct: Actions Universally Justified by Partial Dynamic Policies	[EMvB24]	FORTE'24	Published	1st	basis of [EMvB25a]
• JustAct+: Justified and Accountable Actions in Policy-Regulated, Multi-Domain Data Processing	[EMvB25a]	LMCS'25	Submitted	1st	basis of Chapter 6

Figure 1.1: Articles underlying the contributions in Chapters 2 to 6.

Artefact Title	Citation	DOI and Hyperlink	Thesis Relevance
eFLINT Transpiler to Clingo and Experiments	[MEvB25]	10.5281/zenodo.15188959	impl. & experiments, basis of Chapter 2
SEASO Interpreter and Case Study	[Est25]	10.5281/zenodo.15302471	impl. & case study, basis of Chapter 3
Slack Interpreter	[EMvB25a]	10.48550/arXiv.2502.00138	basis of Section 4.5.3 and used in Section 6.5.2
Reowolf 1.0 Project Code and Documentation	[EH24]	10.5281/zenodo.3559822	subject of Chapter 5 presented in Section 5.2
Reowolf Formalism	[ELH25]	10.5281/zenodo.14936561	definitions & proofs, basis of Chapter 5
JustAct Prototype and Experiments	[EMvB25a]	10.48550/arXiv.2502.00138	impl. & experiments, basis of Chapter 6

Figure 1.2: Artefacts underlying the contributions in Chapters 2 to 6.

- Chapter 2 improves the semantics and tooling for the existing eFLINT language, where users incrementally build specifications, *e.g.*, of data exchange systems.
- Chapter 3 defines SEASO, a new eFLINT-like language for *cooperative specification*, where multiple *agents* incrementally develop a shared specification, and use the same specification to control the contributions of their peers. Here we see that the specification’s definition and enforcement become intertwined.
- Chapter 4 formalises the aforementioned cooperative specification problem, and generalises away from SEASO, evaluating it alongside other specification languages: Datalog, Alloy, eFLINT, and Slick (another new language that is based on SEASO).
- Chapter 5 spreads the agents over a distributed system, so they can no longer maintain their overview of the shared specification. The key is to co-design the specification language (PDL) with the distributed system (the Connector Runtime) to make the changing specification enforceable in this context.
- Chapter 6 generalises away from PDL and the Connector Runtime, and generalises the cooperative specification framework to distributed systems. As in Chapter 5, the agents cannot overview the entire specification, but now there *is* no such specification; within agreed limits, agents act on different views of their system. Nevertheless, we show that agents can decide when they have sufficient information to justify their actions, such that all observers necessarily agree. We then recreate an existing medical data exchange system, and highlight the added flexibility and accountability: agents cooperate to change fundamental data exchange agreements at runtime, and to enforce (their peers’) compliance to the specification.

Finally, Chapter 7 concludes by reflecting on the main contributions. We reflect on their role in the larger research efforts in the AMdEX project and in the research group, and we summarise the various avenues for future work.

## 1.4 Narrative Thread through the Chapters

Here, I link the contribution chapters together by laying out a more personal, candid, and informal narrative. This gives me a chance to present the underlying efforts, my motivations beforehand, and the insights I gained along the way. The narrative is summarised visually by Figure 1.3. Naturally, this figure is an incomplete approximation of a subjective reality, but I hope that it helps to paint a picture.

**In the Beginning** Prior to my arrival, research at the university aimed to close the gap between data exchange software systems and the legal regulations that specify how they are permitted to behave. The *FLINT* language was developed as

a means of modelling legal relationships [vDvdSvE16, BvDdB<sup>+</sup>22]. FLINT models relationships between institutional entities by implementing *Hohfeld’s framework* for legal reasoning. For example, each FLINT specification models actions as actor-recipient relationships, and specifies their pre- and post-conditions as constraints over entity attributes. Then *eFLINT* (‘executable FLINT’) was developed as a FLINT variant for automated case-analysis, via a collaboration with CWI, spearheaded by dr. Thomas van Binsbergen [vBLvDvE20, vBKB<sup>+</sup>21]. The *eFLINT* interpreter digests input specifications, enumerates the normative violations in input *scenarios*; for example, an action was triggered while *prohibited*, because its pre-condition was unsatisfied. Somewhere along the line, *derivation rules* were added to the language, internalising the fundamentals of logic programming. For one thing, these rules enabled *qualifications*. For example, when a rule derives *bid* instances from *raise hand* instances, the *eFLINT* specification internalises a notion of *qualification*: the physical event of raising a hand circumstantially *counts as* the social act of bidding in an auction. The language grew and changed to meet the needs of the students and researchers that used it as a modelling and reasoning tool in various application contexts.<sup>4</sup> I joined their ranks under Thomas’s supervision, and I turned my attention to *eFLINT*, asking: how can this approach be adapted in a context where everything about the specified system becomes distributed, decentralised, and federated, to afford its application to regulating data exchange systems?

**Developing eFLINT** Chapter 2 takes the *eFLINT* language as a starting point, approaching its application to the specification of (norms of) data exchange systems. First and foremost, we refine the language by filling holes in its existing definition, such that each *eFLINT* specification has a single, standard interpretation. In the process of this work, we encountered a flaw in the existing *eFLINT* interpreter, concerning a facet of the language that was under-specified. The issue arose in *eFLINT* specifications which express reasoning *by default*; some specifications have behaviour that is unexpected, and sensitive to unspecified details. To close the gap between our new interpreter and the relevant literature, and to minimise our own effort, our definition of *eFLINT* is expressed as a translation to the existing *Clingo* language [GKKS19]. *Clingo* has the appropriate logical foundations [GL88], and its open-source reasoner addressed the problems with reasoning by default. Our translation repurposes the *Clingo* reasoner as the reasoning back-end for a new interpreter of our *eFLINT* variant. After an experimental comparison of the two interpreters, we conclude that we have correctly redefined and reimplemented the original *eFLINT* semantics, except where we have deviated on purpose, in cases of reasoning by default. As we had hoped, we also found that our new interpreter exploits

---

<sup>4</sup>Somehow, updates to Thomas’s language were most frequent when Thomas was out of office.

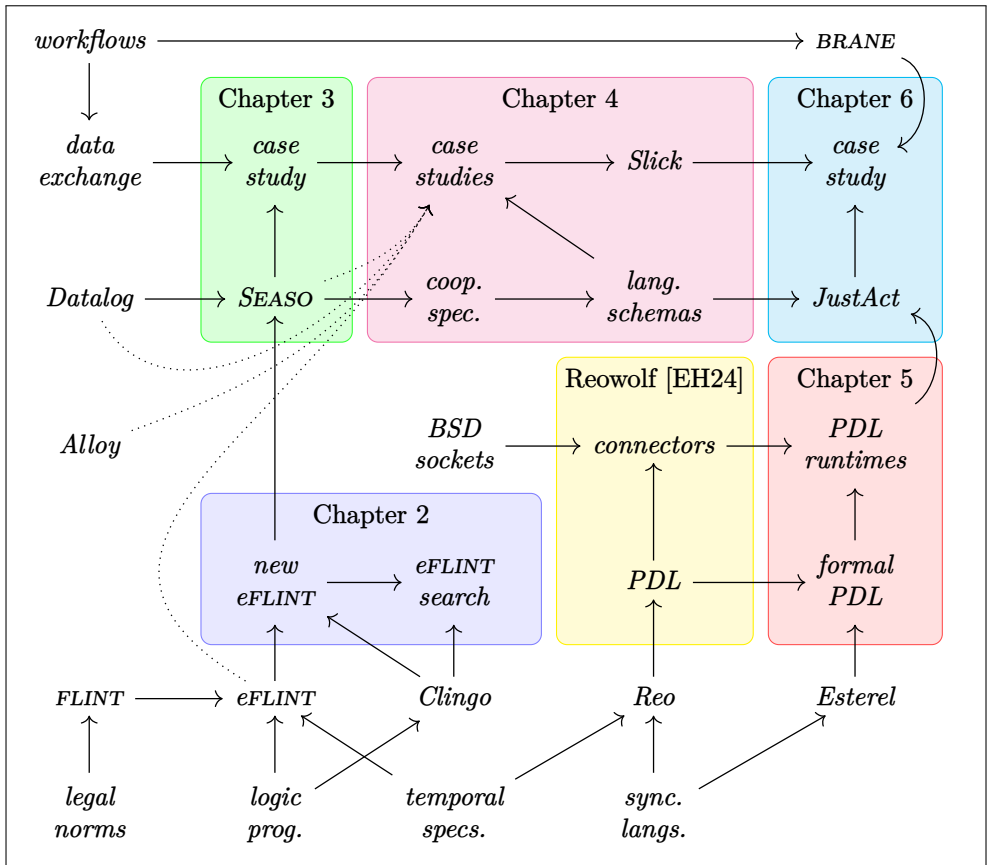


Figure 1.3: Informal overview of tools, languages, frameworks, and research topics relevant to this thesis (italicised text), how they have influenced one another (influencer  $\rightarrow$  influenced), and in which clusters they are presented (coloured rectangles).

the extensive performance engineering behind Clingo to usually achieve significant speedup. Most interestingly, our approach lets us exploit the powerful answer-set solving features of Clingo, affording new usages of eFLINT specifications: users can automate the (bounded) search for (counter)examples to user-defined constraints over scenarios. This usage is common in existing eFLINT-like languages, such as FIEVeL [VC08] and Symboleo [PRR<sup>+</sup>24]. We hope this work to complement the existing applications of eFLINT to data exchange, *e.g.*, to model-check the logical consistency of normative data sharing agreements.

From Chapter 2, we take the lesson that logic programming is fertile ground for a declarative expression of many facets of our systems. It is easy to specialise the interpretation of logical variables by their syntactic structure, and this suffices for us to model many concepts at layers that seem separate at first glance. For instance,

in Chapter 2, our eFLINT translation to Clingo embedded eFLINT’s notions of *truth* alongside *permissions*, *duties*, and *events*, which are associated with radically different modalities, and so the corresponding concepts are usually strictly separated. We didn’t stop there; in Chapter 2, we even embedded a significant chunk of the rules eFLINT semantics (defining eFLINT) at the meta-level as Clingo rules alongside the Clingo rules translating the domain-level (using eFLINT). We observed that eFLINT adopted a characteristic of the logic programming paradigm which is particularly desirable in our context: semantic objects that are the focus of users are highly explainable, because they arise entirely from reasoning steps formulated in terms of program rules under user control.

**Branching Off** Chapter 3 is the result of a language-design process, reflecting my attempt to build a lightweight language that is inspired by eFLINT, but which tightly focuses on my use case: the incremental specification of complex, inter-domain cyber-social systems. The result was SEASO, a very simple domain-specific language. Like eFLINT specifications, SEASO programs 1. model systems as a collection of named relations, denoted by a collection of nested, typed tuples, 2. fundamentally distinguish domain elements representing *truths* from *violations*, and 3. are extensible, affording the alteration and refinement of models. Put together, SEASO affords the same essential usage of eFLINT: programs are incrementally extended, updating a model of the domain of discourse. The presence of explicit *violations* affords SEASO’s role as a specification language: each program specifies that its extensions should add no violations. However, SEASO differs from eFLINT in some key points.

1. The inbuilt integer arithmetic operators and semantics are removed.
2. The syntax is decoupled from legal concepts like *duty-holder* and *action-recipient*.
3. eFLINT’s expression combinators are replaced with Datalog-style rules [MTKW18].
4. The stable-model semantics underlying the logical inference of eFLINT and Clingo is replaced by the (related) *well-founded semantics* for logic programming [VGRS91].
5. The novel *seal* construct is added to control how programs are extended.

Points 1–3 all reflect an extreme emphasis on *minimalism* in the language definition, with the goal of affording rapid iteration of the language design, and the rigorous definition and analysis of the SEASO language and its programs. Points 1 and 4 reflect a new emphasis on a *semantic robustness* that is necessary in our context, where models of the system are expected to be constructed piecewise, as norms and details take shape by passing through the hands of many stakeholders, with various specialisations, over long periods of time. More precisely, Point 1 lets us prove that, unlike eFLINT, every finite SEASO program has a finite model, which is computable

in finite steps, and which prescribes unique answers to every conceivable semantic query. So, for example, malicious stakeholders cannot craft SEASO programs which cause the interpreter to hang. Point 4 subtly changes the meaning of programs which (perhaps unintentionally) express inconsistent logical properties. For example, the original eFLINT interpreter erroneously answers *true* to ‘is Amy a user?’ given the specification ‘Bob is a user but Amy is a user only if Amy is not a user’, while our new eFLINT interpreter answers ‘your specification is inconsistent’ and our SEASO interpreter answers ‘Bob is a user but Amy’s status is inconsistent’. Intuitively, SEASO models ‘contain’ logical inconsistencies, rather than collapsing entirely. The final Point 5 is a very modest technical contribution: like visibility keywords in Java or type-declarations in Haskell, it lets SEASO programs control the usage of program constructs, quickly raising errors when programs are used improperly. However, Chapter 3 extensively explores the potential of this language feature as an additional meaningful layer of SEASO specifications. In a nutshell, like eFLINT’s normative violations, these new *seal* constructs in SEASO programs internalise user requirements that must be expressed at some point: which *changes* to the program itself are permitted? For example, are data-consumer stakeholders permitted to alter the legal experts’ definition of legal consent? Probably not. But are they allowed to add and remove particular instances of consent? Probably in some circumstances. In Chapter 3, we build a suite of reusable interdependent SEASO specifications which precisely define and relate many concepts from the data exchange literature, including events, archetypes,<sup>5</sup> workflows, and access control policies.

The work behind Chapter 3 was important to my personal development; working on and with SEASO clarified ideas that persisted throughout my future works, which is evident in the subsequent chapters. Unfortunately, the paper itself was never accepted for publication, but the reviewers of our rejected drafts directed me to relevant work on topics such as *argumentation frameworks* and the *Alloy specification language* [Jac03]. Eventually, I became more interested in the ideas behind SEASO than in the language itself, and so I focused on exploring these ideas instead.

**Zooming Out** Chapter 4 generalised the ideas of Chapter 3 beyond SEASO. First, we finally precisely formulated the problem of *cooperative specification*: where autonomous agents incrementally build a shared specification that captures everyone’s contributions. Crucially, our framing emphasised the need for control over the process. How can agents avoid (unintentionally) stepping on each others’ toes? We took the approach of internalising the means of control in the shared specification itself. Chapter 4 provides *language schemas* and *control triples* as theoretical tools for

---

<sup>5</sup>SEASO was born from attempts to formalise these archetypes in eFLINT alongside Nina Verheijen, who was working on her Master’s thesis at UvA at the time. We struggled with the lack of control.

precisely defining and evaluating languages for solving the cooperative specification problem. Through this lens, Chapter 4 looks back on the SEASO language, and situates it in the vast space of possible approaches, languages, and cooperation scenarios. We evaluate our framework and perform a cursory exploration of the language space by adapting languages Datalog, eFLINT, and Alloy for cooperative specification. We identify a ‘static–dynamic’ spectrum on which composition-control features fall; in summary, *static* features are easier to evaluate, but have limited expressivity. We re-frame the *sealing* mechanism – developed for SEASO in Chapter 3 – as just one of the possible composition-control mechanisms, situate it on the static end of the spectrum, and explore its use in other languages. We identify *dynamic invalidity* on the opposite end of the spectrum, and explore its strengths and weaknesses. Finally, we introduce Slick as a variant of SEASO that is designed around dynamic invalidity instead of static seals. In a nutshell, Slick sacrifices some of SEASO’s static robustness for an even simpler language definition and greater expressivity.

Chapter 4 became a sort of central lynchpin of my thesis, because it finally clarified our specification-centric approach. Also, it resulted in the Slick language, which would become a useful exploratory tool, and would become integral to the case study in the final Chapter 6. There, the expressivity of Slick afforded our specifications which capture concepts that are particularly complex, because they blur the line between the usual domain- and meta-level facets of the specification. For example, in Chapter 6, Slick programs let legal experts specify that data-consumers are only permitted to create consent (at the meta-level) on the condition that they are known to be the *subjects* of the data (at the domain-level).

**Meanwhile** Chapter 5 introduces the *Reowolf project*, a line of research that ran concurrently with the AMdEX project that drove the previous chapters. Chapter 5 also reflects the paradigms of a different community: formal (synchronous) *network* and *coordination* languages. But over time, this line of research would influence and be influenced by my other research. Chapter 5 reports on just the most relevant facet of my role in developing *Reowolf connectors*, which aimed to replace BSD-style network sockets (like good old TCP and UDP) for multi-party communication sessions on the Internet. Reowolf extended the programming methodology of the *Reo coordination language* to the Internet (hence the name). Reo had been mostly applied to software systems in shared memory, where its highly abstract *protocols* are used to specify dataflow-dependencies between software components. The strengths of Reo come from its extremely abstract relation- and constraint-based formalism, and its design emphasising modularity and compositionality [Arb11]. Reo has seen significant work in developing its Reo-to-Java glue-code generator tool (‘the Reo compiler’ [DA18])<sup>6</sup>

---

<sup>6</sup>In fact, my master thesis [Est19] developed new a Reo compiler back-end, translating to Rust.

and formal methods of systematically reconfiguring Reo protocols and verifying their properties. Reowolf raised the question: can we use a Reo-like language to specify and verify internet protocols, such that Reo’s usual compositionality and safety properties follow? As the core developer, I worked with some dear CWI colleagues for a year to develop Reowolf’s *Protocol Description Language* (PDL), a specialised Reo variant, alongside the *Connector Runtime*, a distributed and incremental PDL interpreter. The idea is that our *connectors* played the role of BSD-style sockets, letting users drive communications by exchanging datagrams with their peers. But the novelty of connectors is their extreme configurability: users control connectors via PDL protocols; programmers can time and synchronize messages in flight between multiple parties. For example, PDL trivialises the expression of synchronous, distributed, and ad-hoc transactions, whose low-level implementations are far from trivial. For example, Amy can specify ‘Send “hi” to Bob or Eva iff Bob sends “hello” to Dan’.

But Chapter 5 does not focus on the design, implementation, and testing of PDL or connectors.<sup>7</sup> Instead, it reports on a more recent effort: we summarise the result of formally 1. defining Reowolf’s PDL, 2. proving key PDL properties, and 3. characterising runtime systems that continuously construct communication behaviour from PDL protocols. Thus, we clarify what exactly the Reowolf project contributed, such that the key ideas can be applied in other contexts. Notably, we distinguish dual notions of PDL behaviour. 1. Users constrain what behaviour is *accepted*, generalising over unknown parts of the session with unknown peers. 2. The runtime system *constructs* behaviour from the protocol, interpreting it as a closed system. We highlight how these notions make PDL fit for its purpose; *acceptance* has the strengths of constraint programming, because syntactically composing PDL protocols precisely intersects the sets of behaviours they accept. We highlight how our application context sets our notion of runtime system apart from similar works, such as the Esterel and Lustre languages. PDL runtime systems are largely decentralised, and protocols can be changed on the fly. For example, consider the contrast to Esterel, whose specifications are used to generate runtime systems in overview first, which generally cannot change once communication has begun.

Work on Reowolf in general, and formally verifying our proofs in Coq<sup>8</sup> [BC13] in particular, were formative in my more methodical thinking and communicating about formal systems in general and specification languages in particular.<sup>9</sup> Over time, the essential tenet underlying Reowolf merged with the main idea behind cooperative specification. I focused on the potential in – and challenges of – framing the fuzzy

---

<sup>7</sup>The Reowolf deliverables [EH24] detail these, but steel yourself for another 150+ pages.

<sup>8</sup>Since this work, Coq was renamed to Rocq, after decades of research and history. Personally, I find the original name charming, but I suppose I can see why it was changed.

<sup>9</sup>If it were up to me, stakeholder negotiations would be conducted via specifications in Coq.

problem of eliciting and combining multiple agents' requirements on their shared system as a continuous specification problem. The first step to productivity was to choose the right language for expressing the requirements. Chapter 5 convinced me that the composite specifications that result from this process can be executed automatically and reliably, even in (mostly) decentralised runtime systems.

**Putting it all Together** On the one hand, Chapter 6 makes a contribution that best reflects the earliest research effort during my time as a PhD student: addressing the decentralised (re)definition and enforcement of complex normative policies. Chapter 6 reflects what I intended from the start, *e.g.*, in my poster for the NWO ICT Open conference in 2022 (Figure 1.4). On the other hand, Chapter 6 leverages more recent results shown in the other chapters. Much as Chapter 4 generalised the problem and SEASO language in Chapter 3, Chapter 6 provided a generic framing which combined the continuous, cooperative specification in Chapter 4 with the decentralised runtime system context of Chapter 5, *e.g.*, generalising away from PDL and the Connector Runtime. In a nutshell, in Chapter 6, we characterise the languages *and* runtime systems that afford meaningful cooperative expression and enforcement of specifications. We define the *JustAct* framework as an abstract ontology, a collection of key requirements, and a specification of what happens when these requirements are not met. The ontology and requirements characterise and inter-relate the chosen 1. policy language, expressing composable normative (meta)specifications, 2. distributed runtime system, which mediates inter-agent communications, 3. agent *statements*, which contribute to the policy, and 4. agent *actions*, which are regulated by the policy. By remaining abstract, the JustAct framework itself remains extremely generic, primarily prescribing the responsibilities on the agents to regulate their own behaviour and the behaviour of the peers. The framework leaves room for different kinds of systems, and preserving the autonomy of the agents to act as they please. But a key concept – reflected in its name<sup>10</sup> – is that JustAct fundamentally supports the task of auditors: checking which *actions* are *justified* by their policies, *i.e.*, permitted. This is despite the decentralisation of the agents and their communications. We demonstrate the applicability of the JustAct framework via a case study; we instantiate our framework and reproduce the use cases of the BRANE medical workflow processing system [VCB21, KAA<sup>+</sup>24]. Moreover, we show how our version of BRANE supports new use cases. For example, in BRANE today, the organisation-local policies regulating the *authorisation* of workflow tasks are strictly separated from BRANE's inter-organisation control plane, to prevent sensitive medical (meta)data from leaking out of local policies. But because our prototype uses the same notion of policies for inter- and intra-organisational policy,

---

<sup>10</sup>Credit for the name goes to Thomas. I must admit that he is better than me at naming things.

# Norm-Conformant Data Exchange Between Distributed Agents

Christopher Esterhuysen  
University of Amsterdam // Complex Cyber Infrastructures

c.a.esterhuysen@uva.nl

## The AMdEX Project

**Context:** Organizations are eager to come together to share and exchange data and compute resources.

**Problem:** It is difficult to exchange data while preserving relevant norms such as EU regulations and access control policies.

**Solution:** Build an infrastructure for generic data exchange, programmable with norm specifications to be automatically enforced.

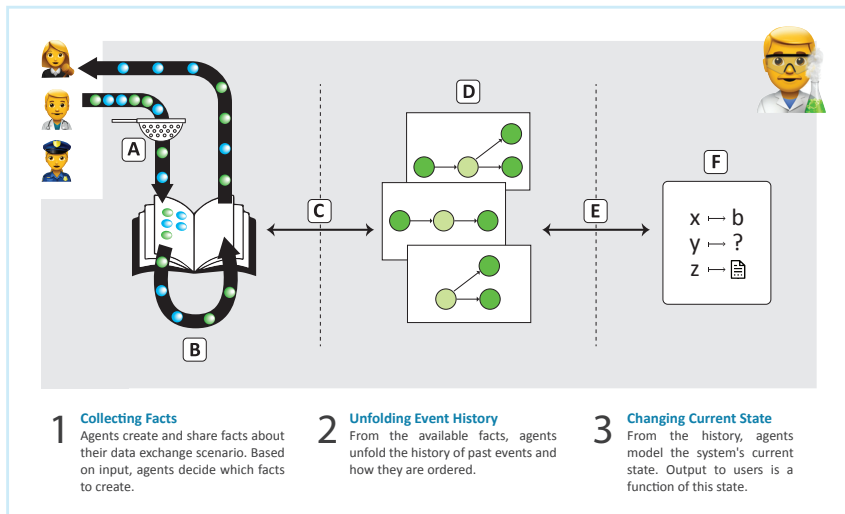
## My Contribution

**Context:** Work exists to formally specify norms, and to drive conformant execution given a specification.

**Problem:** Existing work encourages centralized design, emphasizing consistency over agent autonomy, which does not suit all use cases.

**Solution:** We develop a formalism and prototype which enables balancing consistency and autonomy as best suits the use case.

## Agent-Local Execution



## Related Work

Agents accept facts from peers, but only if they conform to the **static specification**, e.g., they are appropriately **cryptographically signed**.

**A** Facts are created via **inference** based on **multi-modal logics** such as **epistemic logic** and **fluent calculus**.

Event histories are unfolded, but only those that preserve **static safety** properties, defined offline via **model checking**. These choices determine the balance between inter-agent **consistency, autonomy** and **trust**.

**C** Events are partially chronologically ordered, much like tasks in **computational workflows**, encouraging **parallelism** in computing the current state.

The system state is modeled by two abstractions interfaced by unfolding graphs of immutable events, much like the unfolding chains of immutable transaction blocks in **blockchains**.

**E** Manual and automated **monitoring** of the facts, histories, and state incentivize agents to play by the rules.

**Acknowledgements:** This work is part of the AMdEX Fieldlab project supported by Kansen Voor West EFRO (KVVW00309) and the province of Noord-Holland.



Figure 1.4: My poster for the 2022 edition of the NWO ICT Open Conference.

agents can benefit from choosing to share non-sensitive parts of their local policies. As in Chapter 3, this showcases a pervasive idea: there is benefit in situationally *internalising* requirements in the shared specification; the contributor experiences it as control over their peers, while their peers experience it as insight into the requirements, leading to better informed decisions.

In Chapter 6, once again, multiple agents incrementally specify the behaviour of their system. The specification languages themselves have changed little since Chapter 3, but the new framing gives them new significance. Like Chapter 5, JustAct relaxes agents' views on each others' specifications, such that they can work concurrently, and keep sensitive (meta)data to themselves. Unlike Chapter 5, the nature of their synchronisation is configurable also. We show that statically agreeing which details must be synchronised suffices to create complex and configurable inter-agent power dynamics, just by selecting specifications as usual. Thus, even simple specifications play complex roles. Fortunately, this complexity is curbed by our unifying view on language semantics; the same specification-evaluation function is the basis of many important activities that are often distinguished: auditing, runtime monitoring, action planning, and model checking. The key question is always the same: given *this* specification is *that* behaviour acceptable? Naturally, the implementations and case study in Chapter 6 is just the tip of the iceberg. Many combinations of language and configuration remain to be explored and evaluated. Which other systems can we recreate and improve?

## eFLINT Semantics and Model-Checking via Clingo

### Abstract

Since its introduction at GPCE2020 [vBLvDvE20], the eFLINT norm specification language has been used in academic and industrial applications to specify and automate compliance for various norms, such as privacy regulations and data processing agreements. The eFLINT interpreter has been used to automate the analysis of real-time or historical cases by computing logical consequences and reporting normative violations.

To support future language and tooling developments, we contribute a formal definition of the language as a translation to first-order logic programming with stable model semantics. The described semantics aligns with the previous semi-formal descriptions of the language, but resolves issues relating to logical inference with negative antecedent and aggregation operators. Specifically, we formalise the connection between eFLINT’s derivation rules and Horn-clauses under the stable-model semantics. Secondly, by repurposing the Clingo answer-set solver as a highly-optimised eFLINT interpreter, we extend the tool-set for eFLINT with model-checking abstract properties in addition to case-analysis.

We evaluate the new semantics and interpreter via an empirical comparison of the existing implementation to our prototype implementation. We observe that the expected subset of our tests have equivalent behaviours. Moreover, we find that our use of Clingo for reasoning causes our new interpreter to generally outperform the original interpreter.

**Basis of this Chapter** This chapter adapts the GPCE2025 conference article [EMvB25b] and adopts its supplementary artefact, which is available at [MEvB25]: a new eFLINT interpreter implementation and our experimental data and results. This chapter extends the article with extra explanations and benchmark results.

## 2.1 Introduction

The eFLINT language, first introduced at GPCE2020, is a domain-specific language (DSL) designed to specify and reason about interpretations of normative documents such as laws, regulations, and contracts [vBLvDvE20]. eFLINT has been used in research of normative multi-agent systems [ZMKvE22, PvBSvE22], distributed data processing systems [LSvE20, vBKB<sup>+</sup>21, SMO24, EvB24], and to develop partially automated data governance solutions [AKAA<sup>+</sup>24, vBORS<sup>+</sup>24]. In these cases, eFLINT has been used to specify, for example, (parts of) data protection regulations, data processing and consortium agreements, and data access or usage conditions.

From the start, eFLINT was designed to bridge the gap between the normative concepts typically expressed by legal experts, and event-driven software systems that may require legal regulation. Thanks to its high level of abstraction, the language serves as a *lingua franca* for normative concepts in separate domains. For example, by their common encoding in eFLINT, smart contracts, data protection regulations, and organisational policies have meaningful compositions [vBKB<sup>+</sup>21]. The *eFLINT interpreter*, available online,<sup>1</sup> checks the compliance against an eFLINT specification of given *scenarios*, which instantiate specifications by laying out the events that happen(ed) in the scenario. Precisely, the tool computes logical consequences and enumerates the normative *violations*: unmet obligations and prohibited actions.

Section 2.2 explains the eFLINT language in more detail. Until then, we exemplify the typical usage of the language; the eFLINT interpreter monitors the compliance of a software system by checking system traces (modelled as scenarios) for compliance against regulatory norms (modelled as a specification). Example 2.1 introduces our running example of such a usage; in this case, the specification models the institutional relationships between *users*, *datasets*, and *instants*, and formalises the norm that users must inform data controllers of (the purpose of their) data accesses within a given time frame. The scenario lays out a concrete sequence of events.

**Example 2.1** (Running Example eFLINT Usage). A system is regulated by the following specification, which expresses: accessors of datasets must notify the dataset’s controllers (or the administrator by default) within ten time steps of the access.

```
// Introduce three basic record-types.
// Whenever controls relates controller X and dataset Y, derive existence of X and Y.
Fact instant Identified by Int
Fact user    Identified by String Derived from (Foreach controls: controls.controller)
Fact dataset Identified by String Derived from (Foreach controls: controls.dataset)
```

<sup>1</sup>The Haskell implementation sees active development. But we focus on the version authored on the 13<sup>th</sup> of February 2025: <https://gitlab.com/eflint/haskell-implementation/-/blob/cfe73e9a0ed594e384027ea29529da62dab1fe0f>. See Section 2.9 for details of our dependency on this implementation.

```

// Introduces type-aliases. E.g., any variable 'controller' has type 'user'.
Placeholder controller For user
Placeholder deadline For instant

// Introduce (mutable) relations over users, instants, and datasets.
Fact elapsed Identified by instant
Fact controls Identified by controller * dataset
  Derived from (Foreach dataset: controls(user("Admin"),dataset)
    Where Not(Exists user: user != user("Admin") && controls(user,dataset)))

// Actions are facts that can be triggered (with defined effects).
Act access Actor user Related to dataset, instant
  Derived from (Foreach user, controls, instant: access(user, controls.dataset,
    instant))
  // When instance access(user,dataset,instant) is triggered, create these instances:
  Creates (Foreach controls:
    must_notify(user, controls.controller, access(user,dataset,instant), instant + 10)
    Where controls.dataset == dataset)

// Duties are facts that are violated (under defined conditions).
Duty must_notify Holder user Claimant controller Related to access, deadline
  Violated when Holds(elapsed(deadline))

Act notify Actor user Recipient controller Related to must_notify
  Derived from (Foreach must_notify:
    notify(must_notify.user, must_notify.controller, must_notify))
  // when notify(user,controller,must_notify) is triggered, remove this instance:
  Terminates must_notify

```

The following eFLINT scenario models a hypothetical system configuration just after user Bob accesses the X-Ray dataset. It applies five effects in sequence.

```

+dataset("X-Rays"). // create
+controls(user("Amy"),dataset("X-Rays")). // create
+instant(9). // create
+user("Bob"). // create
access(user("Bob"), dataset("X-Rays"), 9). // trigger

```

Given the above input, the eFLINT interpreter emits the following output. The first lines confirm the input. The final lines report that Bob's access action was *enabled* (*i.e.*, permitted), and it had some effects: Bob acquired the permission and duty to notify Amy and the administrator of the access event.

```

New types: access, controls, dataset, elapsed, instant, must_notify, notify, user
+dataset("X-Rays").
  |-> +user("Admin")
  |-> +controls(user("Admin"),dataset("X-Rays"))

```

```

+controls(user("Amy"),dataset("X-Rays")).
  |-> user("Amy")
+user("Bob").
+instant(9).
  |-> +access(user("Admin"),dataset("X-Rays"),instant(9))
  |-> +access(user("Amy"),dataset("X-Rays"),instant(9))
  |-> +access(user("Bob"),dataset("X-Rays"),instant(9))
executed transition: access(user("Bob"),dataset("X-Rays"),instant(9)) (ENABLED)
  |-> +notify(user("Bob"),user("Admin"), ...,instant(19))
  |-> +must_notify(user("Bob"),user("Admin"), ...,instant(19))
  |-> +notify(user("Bob"),user("Amy"), ...,instant(19))
  |-> +must_notify(user("Bob"),user("Amy"), ...,instant(19))

```

The eFLINT language is motivated, described, and demonstrated in research articles [vBLvDvE20, vBKB<sup>+</sup>21]. The syntax and fundamentals of the semantics are defined using conventional mathematical notation and precise natural language. For example, [vBLvDvE20] makes clear that each specification denotes an action-labelled transition system, and that each scenario denotes a sequence of actions that traces a path through the transition system. Figure 2.1 visualises a scenario, its constituent actions, and their corresponding transitions. The existing eFLINT interpreter agrees with the existing description of the eFLINT semantics. For example, the details in [vBLvDvE20] suffices for users to agree that **Derived from** (**Foreach** dataset: ...) in Example 2.1 denotes an eFLINT *derivation clause* which expresses ‘the administrator control of datasets which have no other controllers’.

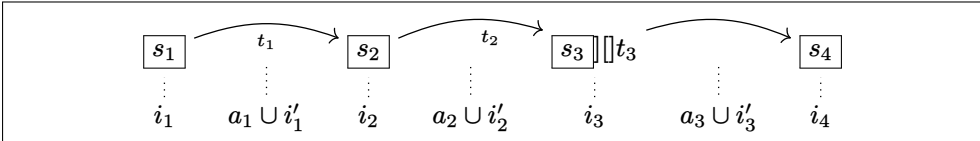


Figure 2.1: A trace through states  $s_{1-4}$  over transitions  $t_{1-3}$ , induced by a scenario triggering actions  $[a_1, a_2, a_3]$  in sequence. Each state and transition models the (current) normative relationships as (elements of) some structured data base  $i$ .

However, at the time of writing, facets of the eFLINT semantics are not rigorously defined. Consequently, details of the semantics remain unclear. For example, does the administrator retain control over the X-Rays once Amy has taken control, *i.e.*, after the events of the running example scenario? In the original interpreter, querying `?controls(user("Admin"),dataset("X-Rays"))` after Example 2.1 yields true, so the answer seems to be yes. However, this observed behaviour conflicts with the plausible interpretation of [vBLvDvE20], that a derivation clause should yield a fact precisely when all of its conditions are satisfied. Then why is the administrator’s control of the X-Rays derived in a state where another user (Amy) already has control? In

general, we observe that the behaviour of the eFLINT interpreter deviates from our expectations in interpreting negation (**Not**), iteration (**Foreach**) or aggregation (*e.g.*, **Max**) expressions. But without a formal eFLINT semantics, how can we distinguish surprising but intended behaviour from bugs in the interpreter implementation? In either case, the current behaviour is undesirable, because its evaluation of expressions is sensitive to semantically irrelevant details, such as the alphabetic ordering of type-identifiers. For example, the outcome of the query is flipped if `user` is named `agent` instead.

Presently, we contribute a (re)definition of the eFLINT language with a new semantics. Our goal is to formalise eFLINT’s semantic concepts in detail, such that eFLINT scenarios and specifications have unique and unambiguous meanings. We give the semantics an intuitive and mature foundation: first-order logic programming under the stable-model semantics [GL88]. This is also pragmatic choice, as it lets us leverage existing logic programming tools. Precisely, we formulate a translation of eFLINT specifications and scenarios into (the input language of) *Clingo*, the answer-set solver. Thus, eFLINT inherits the clarity of Clingo’s stable model semantics, and we repurpose the Clingo solver as the main component of our eFLINT interpreter.

We recreate the existing eFLINT descriptions and interpreter as much as possible. Specifically, we realise the description in [vBLvDvE20] by aligning key eFLINT constructs with the inference rules (Horn-clauses) of logic programming. Thus, users are afforded powerful, (de)compositional reasoning about scenarios and systems via specifications. As part of this (re)design, we make two changes relative to the existing eFLINT interpreter. Firstly, the ambiguity of the original description regarding the semantics of derivation clauses with negation, iteration, and aggregation is resolved. We do so by connecting these constructions to the literature on (the challenges of) reasoning with negated conditions in logic programming. Consequently, in Example 2.1, the specification expresses ‘in each state, each dataset is controlled by the administrator unless it has no other controller’, and then, as expected, the administrator’s default control over the X-Rays is removed as Amy takes control. Secondly, we leverage our translation to Clingo to support a new, more general use case for eFLINT: via Clingo, our tool *searches* for scenarios with user-defined properties. For example, ‘find all counterexamples to the proposition: scenarios taking only permitted actions violate no duties’. This use case has many practical applications, *e.g.*, in model-checking and recommendation, and is already popular with languages related to eFLINT, such as Symboleo [PRR<sup>+</sup>24] and FIEVEL [VC08].

We formulate our translation from eFLINT to Clingo as the multi-stage translation pipeline shown in Figure 2.2. This helps to structure our presentation. But moreover, it introduces novel intermediate representations which are useful in their own right. For example, we introduce *Core eFLINT* as a new language between eFLINT and Clingo;

although it is entirely embedded in Clingo, it captures much of the eFLINT semantics, and it unifies eFLINT specifications with eFLINT scenarios.

$$\boxed{\textit{specification} \times \textit{scenario} \Longrightarrow \textit{Core eFLINT} \longrightarrow \textit{state traces} \longrightarrow \textit{Clingo}}$$

Figure 2.2: Translation from an eFLINT specification-scenario pair to Clingo via the intermediate eFLINT core and state trace abstractions. Each ( $\Longrightarrow$ ) is implemented via an explicit translation function, while each ( $\longrightarrow$ ) is implemented as an embedding of the source language in the target language, *i.e.*, a library in the target language.

Concretely, we make the following contributions:

- We define a translational semantics for eFLINT specifications and scenarios with first-order logic programming and stable-model semantics as the semantic domain.
- We implement our semantics by using the Clingo solver for reasoning.
- We define Core eFLINT as a language embedded in Clingo, capturing the essence of eFLINT: the representation and reasoning of stateful facts and violations.
- We experimentally evaluate the prior contributions via a suite of test specification-scenario pairs, confirming the correctness – and benchmarking the performance – of our new eFLINT interpreter, in comparison to the existing interpreter.

Our artefact at [MEvB25] includes our new interpreter implementation, our experimental tests and results, and the tools needed to reproduce our evaluation.

This chapter proceeds as follows. Sections 2.2 and 2.3 begin by giving our own accounts of the existing eFLINT and Clingo languages, the source and target languages of our translation, and their respective background literature. Sections 2.4 to 2.7 build up the stages of our translation pipeline ‘bottom-up’, *i.e.*, we build layers of abstraction atop Clingo. The stage in Section 2.7 handles the two forms of reasoning: scenario-compliance and scenario-search. Section 2.8 briefly describes the implementation before Section 2.9 evaluates its behaviour and performance in comparison to the original eFLINT interpreter. Finally, Section 2.10 identifies related and future work, before Section 2.11 concludes with a summary.

## 2.2 Translation Source Language: eFLINT

**Normative Specification Languages in General** Each (*formal*) *specification language* affords the systematic development of complex artifacts called *specifications*. The rigour of a formal language definitions is useful because it ensures that each specification has a precise, unambiguous meaning. Automated analyses can then reveal or confirm (un)desirable properties of specifications and the concepts they

model. For example, a formal specification of a communication protocol can be used to regulate a distributed system; events that deviate from the specification can be ruled out beforehand, monitored, or audited via the inspection of system logs.

*Normative* specification languages are designed to model norms (*e.g.*, laws and contracts) that regulate social systems. These languages formalise notions of *permission*, *prohibition*, and *obligation* and facilitate the systematic modelling and reasoning about cases. Languages based on the Hohfeldian legal framework [Bia15, Dd16], such as Symboleo [PSA<sup>+</sup>22, SPA<sup>+</sup>20] and eFLINT [vBLvDvE20, vBKB<sup>+</sup>21] are action-oriented, by formalising the mutable relations between institutional entities: their *powers* and *duties* to act. The focus on actions affords the specification and control over software systems' interactions with the social world. Such systems are also beholden to norms; *e.g.*, systems processing personal data may be subjected to the *General Data Protection Regulation* of the European Union [Eur16].

**The eFLINT Language** Here, we give a brief account of the eFLINT language constructs and how they afford the modelling of (normative) domains of discourse. Throughout, we demonstrate concepts with the running example Example 2.1.

Each specification lays the ontological foundations of the model as a collection of record-type definitions. For example, `controls` is a record with `controller` and `dataset` fields. Each type is also defined alongside a collection of *clauses*, which *instantiate* these types, identifying particular (relationships between) entities. For example, `controls(agent("Amy"), dataset("X-Rays"))` is an instance of `controls`. The specification models the system by building up a collection of *current* instances. However, instances never occur alone; they are always contextualised by an *attribute*, such as *holds* or *actViol*. Users of the language (statically) agree on how these attributes are interpreted as modalities. For example, instances with *actViol* witness particular cases of *violating actions*, because they were not permitted when they were triggered. Later, we see how the eFLINT semantics formalises such meanings by *inferring* some attributes from others. For example, by definition, instances with *created* also have *holds*. Users model systems by defining clauses like (`Derived from ...`) and (`Terminates ...`), to capture different normative concepts; precisely, they also infer attributes from others. For example, associating `Derived from (Foreach user...)` to the `access` type, the user associates *derived* to `access(-type)` instances. Clauses are evaluated in the context of a concrete *current* state. The states only become concrete when the specification is evaluated alongside a *scenario*. The scenario lays out the instance-attribute pair which gets each successive state started. Typically, the attribute is *trigger*, such that the instance identifies an action that happens in transition to the next state. For example, because `access` is marked `Act`, its instances can be triggered. Some attributes

have effects that persist over subsequent states. For example, when triggered, `access` instances *create* new `control` instances, which persist until they are *terminated*.

Thus, eFLINT models complex and stateful cyber-social systems, unambiguously specifies which states are (not) desirable, and lays out the details of concrete scenarios.

## 2.3 Translation Target Language: Clingo

Recall Figure 2.2: we ultimately define the semantics of eFLINT by translating arbitrary eFLINT specifications and scenarios to *Clingo programs*: sets of Clingo rules.

### 2.3.1 Clingo in Context: Logic- and Answer-Set Programming

The *logic programming* paradigm operationalises formal logics: logical theories correspond to declarative logic programs, and the evaluation of logical formulae corresponds to executing the programs. Different logic programming languages and tools are designed for different cases, striking different compromises between expressivity, performance characteristics at runtime, guarantees about program termination, and so on. [CGT89] overviews the work and tools around the *Datalog* language, and [Pac24] is a recent PhD thesis focusing on Datalog.

*Answer-set-programming* or *-solving* can be seen as sub-paradigm of logic programming with an emphasis on modelling search problems via rich language features. Its name comes from its basis on the *answer-set semantics*, which is also called the *stable-model semantics* [GL88]. This semantics is characterised by attributing a set of *stable models* to each program; each is interpreted as a solution to the search problem. [GKKS22] studies the formulation of search problems in these languages. For example, there are many different ways to encode the same search problems in answer-set programs, and the choice of encoding affects the efficiency of reasoning.

*Clingo* is an answer-set solving language and toolset [GKKS19]. It is well-known in the answer-set solving community for its maturity and its high performance in solving. For example, [CGMR16] reports on the results of an answer-set solving competition where various Clingo components outcompeted similar tools.

### 2.3.2 Features of The Target Fragment of Clingo

Here, we explain the features of Clingo that are the target of our translation. Section 2.10 discusses the potential to target other languages (some of) these features.

We pervasively use Clingo’s **general logic programming** features, *e.g.*, these are shared with Datalog. Each rule asserts that the truth of a *consequent* term (or ‘conclusion’) is implied by the conjunction of listed *antecedent* terms (or ‘conditions’).

Each rule denotes a set of *concrete rules* for all substitutions of the variables. *E.g.*, `wet(Day) :- rainy(Day), cold(Day)` expresses that ‘rainy and cold days are wet’. The `:-` can be omitted from rules with no conditions. Ultimately, Clingo maps programs of this form to (*logical models*). Each model prescribes a Boolean truth-valuation to each *term* (also called *atom*) that is variable free (also called *ground*). For example, given the above rule as input, Clingo outputs the empty model; `wet(Day)` is not valuated because it contains variable `Day`, and `wet(monday)` is valuated to *false*, because it is absent from the model, *i.e.*, no concrete rule inferred it. But if the same input is extended by the condition-free rule `cold(monday)`, the output model is `{cold(monday)}`, which attributes truth to the term `cold(monday)` but falsity to the term `wet(monday)`. The first rule did not apply with `Day = monday`, because `rainy(monday)` is still false.

Clingo admits **negated** antecedents, matching `not a`, expressing *weak* negation or negation *by default* of *a*: the inability to infer the truth of *a*. From a proof-theoretic perspective, this promotes the unprovability of *a* to a proof of  $\neg a$ . This is distinct from *classical negation* or *strong negation*, which must be proven independently of *a*. Clingo denotes classical negation of each term *a* as `-a`, but we make no use of it. Weak negation affords *reasoning by default*: drawing conclusions from the absence of truth. *E.g.*, `wet(Day) :- rainy(Day), not sunny(Day)` expresses ‘rainy but not sunny days are wet’. Languages supporting weak negation must address the absence of *canonical* (*i.e.*, unique) semantic interpretations for syntactically valid rules such as `p :- not p`. Many solutions have been explored in the literature. Some languages opt to simply disallow these programs. Answer-set solvers like Clingo generally embrace the lack of a unique interpretation and simply enumerate all the separate *stable models* as alternative answers. Intuitively, this enumerates the possible combinations of assignments to logical variables (‘models’) that satisfy each program rule. For example, `p :- not p` has zero stable models as none satisfies this contradictory rule.

Clingo’s answer-set semantics supports the expression of (**integrity**) **constraints** over the values of logical variables. These take the form of rules with no explicit consequent, which express the conditional truth of  $\perp$ : logical inconsistency. For example, `:- not wet(monday)` asserts that ‘it is inconsistent to assume that Monday is not wet’ or, perhaps more intuitively, ‘Monday must be wet’. We use such constraints in Sections 2.4 and 2.5 to protect the abstractions in our intermediate representations. Effectively, these rules let us document our semantic invariants. In each stage of our translation, we argue why the constraints of prior stages are satisfied.

Clingo admits (syntactically) **nested atoms**, *i.e.*, the logical variables in the model may be identified by arbitrarily deep syntactic structures. Correspondingly, rule consequents and antecedents admit nested terms, and variables can bind subterms at any depth. For example, the rule `controls(agent("Amy"), dataset(D)) :- dataset(D)`

is admissible in Clingo, but it is inadmissible in Datalog, because the consequent nests the term `agent("Amy")`. In the literature, this language feature is also called *function symbols*; reflecting the distinction between the *function symbol* constants of nested terms (e.g., `agent` above) in contrast to *predicate symbols* (e.g., `controls` above); only the latter construct terms that are valuated by the model. Some formalisms emphasise their separation by drawing function and predicate symbols from different alphabets, but like eFLINT and Clingo, we do not. So, for example, in the above rule, `dataset` acts as both a function symbol (in the consequent) and a predicate symbol (in the antecedent). Regardless, the ability to nest terms affords significant modularity in definitions. Consider how `controls(user, dataset)` in Example 2.1 is independent of the internal syntactic structure of terms matching `agent(...)`.

We exploit Clingo’s convenient **tuple terms**, which effectively have the empty function symbol. E.g., `(agent, "Amy")` approximates `agent("Amy")`, but only in the former term can `agent` bind variables. Note that Clingo forbids tuple consequents or antecedents, so `f(x) :- x` is a syntactically valid Clingo rule, but `(f, x) :- x` is not.

Unfortunately, because Clingo supports the nesting of terms, its inference procedure is generally not guaranteed to **terminate**. This problem has no straightforward solution [dSD94], and we make no attempt to solve it ourselves.<sup>2</sup> Instead, we inherit the behaviour of the current eFLINT interpreter and Clingo alike: the evaluation of some programs simply does not terminate. For example, the Clingo interpreter diverges on input `f(f(x)) :- f(x). f(0)`. Intuitively, these programs denote infinitely large models. In Section 2.6, we address the only threat to the termination of our programs: user-defined eFLINT-derivation and -synchronisation rules.

Like eFLINT, Clingo provides an **integer theory**. Integer operators (e.g., `<`, `+`, `-`) and integer and string constants (e.g., `100` and `"Douglas"`, respectively) are constant symbols given special treatment; the operators are concretely denoted in infix position, and their applications are normalised under evaluation, as one is likely to expect. For example, in Clingo and eFLINT, `3 + 1` normalises to `4` and `0 < 1` holds true.

We often rely on **(in)equality constraints**. These are conditions of the form `a = b`, which assert the syntactic equality of terms `a` and `b` once all variables are concretised. Mostly, we use these to filter assignments to variables. For example, `cool(X) :- person(X) ; not X = "Dan"` asserts that all persons but Dan are cool. Clingo lets `a` be a fresh variable in cases all variables in `b` are bound in other conditions. Thus, `a` can stand in for `b`. For example, `cool(X) :- X = "Amy"` is equivalent to `cool("Amy")`.

Section 2.6 relies on (and discusses and demonstrates) features that make *individual* Clingo rules more expressive. Firstly, we use **conditioned conditions** (or ‘nested

<sup>2</sup>Chapter 3 designs the SEASO language such that inference is terminating, at the cost of other features; notably, SEASO integers are simple constants, because SEASO provides no integer theory.

rules’) such as  $(b : c, d \text{ nested in}) a :- b : c, d ; e$ . Thus we model disjunctive conditions; for example,  $x : \text{not } y$  encodes  $x \vee y$  as the logically equivalent  $x \leftarrow \neg y$ . Note that  $(:)$  and  $(,)$  are the nested counterparts of  $(:-)$  and  $(;)$ , respectively. Secondly, nested rules can be **aggregated** by a preceding `#sum`, `#count`, `#max`, or `#min`, yielding integers. For example,  $f(1) :- 0 = \#count\{x : f(N)\}$  is contradictory.

Section 2.7.2 relies on a feature of Clingo that is atypical outside of answer-set solving: rules admit **disjunct consequents**. For example, rule  $\{a\}$ . *supports* but does not *imply* the truth of  $a$ . There are two stable models: either nothing is true, or only  $a$  is true. We use these rules to encode search spaces of eFLINT scenarios.

## 2.4 State Traces

eFLINT models dynamic (normative) systems as event-labelled transition systems as was visualised in Figure 2.1. The original article [vBLvDvE20] explicitly bases this approach on the *event calculus*, a logical theory for modelling dynamic systems.

### 2.4.1 A Primer on the Event Calculus

Like eFLINT, we are influenced by the situation, event, and fluent calculi. For the curious reader, we give our own brief account of their relation in Appendix A. But here it suffices to focus on the event calculus, which is the main influence.

The event calculus formalises notions of *state*, *fluent*, and their relation as (*holds*)  $in \subseteq \text{fluent} \times \text{state}$ . The fundamental notion of *inertia* is formalised as an axiom, expressing ‘the truth-value of fluents persist from each state to the next until it is acted upon’, where actions and states are (inter-)related by a (partial) ordering on *times*. Particular systems are modelled by naming *actions* and specifying their *effects*: how they change truth in the future as a function of truth at present. Particular scenarios are modelled by causing particular actions to happen at particular times.

### 2.4.2 State Traces, Embedded in Clingo

We capture this fundamental layer of the eFLINT semantics in our most abstract intermediate representation: state traces. As visualised in Example 2.2, each state trace details one scenario as a path through a transition system, representing the linear updates to the system’s current state over time. We adopt a simplistic model, in which many features of the event and situation calculi coincide: the system is unfolded as a sequence of *states*, indexed by a finite prefix of the natural numbers  $\{1, 2, 3, \dots, n\}$ . We use Clingo’s integer theory for traversing and comparing states:  $S+1$  is the successor of state  $S$ , and  $(<)$  orders states. We likewise index the sequence

of transitions, *i.e.*, each  $i$  in  $1 \leq i < n$  identifies the  $i$ th transition from state  $i$  to state  $i + 1$ . We denote the truth fluent  $F$  in state  $S$  as  $in(F, S)$ .

Precisely, Definition 2.1 encodes the syntax and semantics of our state trace language as a Clingo rule set. The first four rules characterise the states: *state* predicates some non-empty and contiguous prefix of the natural numbers. Rule (INERTIA) encodes the main axiom of the event calculus:  $F$  is true in each state where it was added with  $(add, F)$  more recently than it was removed with  $(rem, F)$ . Note that this holds even in cases where the same  $F$  is added and removed simultaneously, in which case the removal suppresses addition (which results in simpler semantic rules than the alternative). Predicates  $in((add, F), S)$  and  $in((rem, F), S)$  have no direct influence on the truth of  $F$  in state  $S$ , but rather affect the truth of  $F$  in state  $S + 1$  onward. It is helpful to think of  $in((add, F), S)$  and  $in((rem, F), S)$  as labelling the *transition* indexed by  $S$ , which goes from state  $S$  to state  $S + 1$ .

Example 2.2 demonstrates the behaviour of state traces in general, by modelling a facet of the running example in particular (Example 2.1): Amy sometimes controls the X-Rays dataset, and the administrator controls datasets that have no other controller. Via Clingo, the state trace acquires the stable model semantics, so our representation of the scenario acquires a unique and unambiguous meaning: per state, the administrator controls "X-Rays" iff Amy does not.

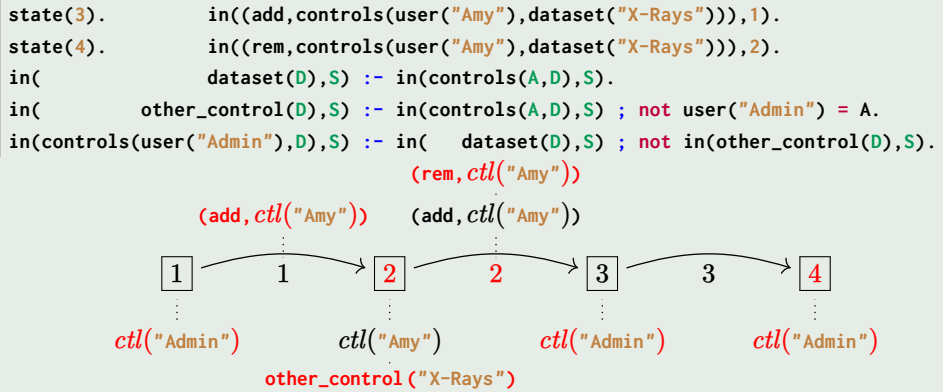
**Notation 2.1** (Semantic Rules in Clingo). As per the tradition of formal languages, when they encode our semantics, we notate Clingo rules in the style of the Gentzen-style sequent calculus or natural deduction. For brevity, we fuse rules with identical antecedents by conjoining their consequents. For example we notate Clingo rules  $a :- f, g(P, Q)$  and  $b(Q) :- f ; f(P, Q)$  fused together as  $\frac{f \wedge g(P, Q)}{a \wedge b(Q)}$ .

**Definition 2.1** (state trace semantics). These Clingo rules assert the contiguity of states in  $\{1, 2, 3, \dots, n\}$  up for some final  $n$ , and encode the *inertia axiom* of the event calculus: each  $(add, F)$  implies the truth of fluent  $F$  until after  $(rem, F)$ .

$$\frac{\top}{state(1)} \quad \frac{in(F, S)}{state(S)} \quad \frac{state(S) \wedge 1 < S}{state(S - 1)} \quad \frac{state(S) \wedge 0 \not< S}{\perp}$$

$$\frac{in((add, F), S) \wedge \neg in((rem, F), S) \wedge state(S + 1)}{in(F, S + 1) \wedge in((add, F), S + 1)} \quad (\text{INERTIA})$$

**Example 2.2** (example Clingo rules defining a state trace). When the following rules are input to the Clingo interpreter along with the state-trace semantic rules (Definition 2.1), the output is as visualised below: the states, transitions, and fluents comprising a particular state trace. We colour each term **red** or black iff it is the conclusion of an example or state-trace semantic rule, respectively. The figure uses  $ctl(X)$  as an abbreviation for the Clingo term  $controls(X, \text{"X-Rays"})$ .



### 2.4.3 State-Internal Rules atop State Traces

The library defined in Definition 2.1 affords the definition of state traces using *add* and *rem* in rules to control the presence and absence of fluents in states. Definition 2.2 defines a syntactic property over rules that ensures that fluents in one state cannot influence fluents in other states. Example 2.3 clarifies the subtleties of this property with some suggestive (counter)examples.

The idea is that *only* Rule (INERTIA) in Definition 2.1 is not state-internal, and all other rules rely on it to relate states. For example, the prior Example 2.2 models (part of) the running example scenario as a set of state-internal Clingo rules which ‘initialise each state’ and ‘inter-relate fluents within each state’, but rely on the state trace semantics (Rule (INERTIA) in particular) to relate the fluents of different states.

**Definition 2.2** (state-internal). Rule is state-internal iff, for each antecedent identical to  $in(F, S)$  for some  $F$  and  $S$ , each consequent is identical to  $in(F', S)$  for some  $F'$ . I.e., if any antecedent ‘reads’ from state  $S$ , all consequents must ‘write’ only into  $S$ . Intuitively, state-internal rules do not let information ‘leak’ out of states.

**Example 2.3** (synthetic examples of (non) state-internal rules).

- $in(f, S) :- state(S)$  is state-internal; no antecedent matches any  $in(F, S)$ .
- $in(x, S) :- in(y, S) ; f(y)$  is state-internal;  $in(x, S)$  is identical to  $in(F, s)$  if  $F = x$ .
- $in'(x, S) :- in(y, S)$  is not state-internal; constant  $in'$  is not identical to constant  $in$ .
- $in(x, 1) :- in(y, S)$  is not state-internal; constant  $1$  is not identical to variable  $s$ .

## 2.5 Core eFLINT

The library defined in this section can simultaneously be seen as a Clingo-embedded domain-specific language (EDSL) and as a core language underneath eFLINT.

Core eFLINT refines the prior state trace abstraction; as before, *fluents* are related to *states* by the *in* relation. But now each fluent  $F$  is an *attribute-instance* pair  $(A, I)$ , *i.e.*, each  $\text{in}((A, I), S)$  attributes  $A$  to instance  $I$  in state  $S$ . Instances are opaque in the semantics, *i.e.*, they are always bound to Clingo variables. Instances are further structured by the users (*e.g.*, by specifications in Section 2.6). Core eFLINT fixes the *attributes*  $\mathcal{A}$  in Definition 2.3, and prescribes their semantics in alignment with the corresponding semantics of eFLINT, as it is described [vBKB<sup>+</sup>21, vBLvDvE20] and with missing details drawn from the Haskell code of the original interpreter.

**Definition 2.3** (instance attributes). This lists the (*instance*) *attributes*  $a, a_1, a', \dots : \mathcal{A} \triangleq \mathcal{A}_t \cup \mathcal{A}_s$ , in states ( $\mathcal{A}_s$ ) and transitions ( $\mathcal{A}_t$ ). Attributes are underlined and **boldfaced** when they occur in antecedents or consequents, respectively, of the core eFLINT semantic rules in Definition 2.4; *i.e.*, they are input and output of the library.

$\mathcal{A}_t$	<u>create</u> , <u>terminate</u> , <u>obfuscate</u> , <u>actTrigger</u> , <b>actViol</b> , trigger
$\mathcal{A}_s$	<b>created</b> , <b>terminated</b> , <b>holds</b> , <b>enabled</b> , <b>dutyViol</b> , enum, <u>derived</u> , <u>suppressed</u> , <u>violated</u>

**Definition 2.4** (core eFLINT semantic rules). The following eight Clingo rules define the semantics of core eFLINT atop the state trace semantics (Definition 2.1).

$\frac{\text{in}(\text{create}, I, S)}{\text{in}(\text{add}, (\text{created}, I), S)}$	$\frac{\text{in}(\text{terminate}, I, S)}{\wedge \neg \text{in}(\text{create}, I, S)}$	(UPPER)
$\wedge \text{in}(\text{rem}, (\text{terminated}, I), S)$	$\wedge \text{in}(\text{add}, \text{terminated}(I), S)$	
$\frac{\text{in}(\text{obfuscate}, I, S) \wedge \neg \text{in}(\text{terminate}, I, S) \wedge \neg \text{in}(\text{create}, I, S)}{\text{in}(\text{rem}, \text{created}(I), S)}$	$\frac{\text{in}(\text{holds}, I, S) \wedge \neg \text{in}(\text{suppressed}, I, S)}{\text{in}(\text{enabled}, I, S)}$	(MIDDLE)
$\wedge \text{in}(\text{rem}, \text{terminated}(I), S)$	$\text{in}(\text{derived}, I, S) \wedge \neg \text{in}(\text{suppressed}, I, S) \wedge \neg \text{in}(\text{terminated}, I, S)$	
$\frac{\text{in}(\text{created}, I, S)}{\text{in}(\text{holds}, I, S)}$	$\frac{\text{in}(\text{holds}, I, S)}{\text{in}(\text{holds}, I, S)}$	
$\wedge \neg \text{in}(\text{enabled}, I, S)$	$\wedge \text{in}(\text{enabled}, I, S)$	(LOWER)
$\frac{\text{in}(\text{actTrigger}, I, S)}{\text{in}(\text{actViol}, I, S)}$	$\frac{\text{in}(\text{violated}, I, S)}{\text{in}(\text{dutyViol}, I, S)}$	

Precisely, Definition 2.4 defines the semantics of Core eFLINT. These rules relate the different attributes of each instance. Note that these rules are all state-internal (Definition 2.2); (only) the UPPER rules in Definition 2.4 use *add* and *rem* to relate

the present and future attributes of each instance. Intuitively, the ephemeral truth of *transition*-attributes (in  $\mathcal{A}_t$ ) like *create* affect the persistent truth of *state*-attributes (in  $\mathcal{A}_s$ ) like *created* and *enabled*. It is helpful to think of each  $in((A, I), N)$  as labelling the  $N^{\text{th}}$  transition whenever  $A$  is a transition attribute, *i.e.*, labelling the transition from the  $N^{\text{th}}$  state to the  $N + 1^{\text{st}}$  state. To follow, in turn, we discuss each cluster of inter-related Core eFLINT rules: { UPPER, MIDDLE, LOWER }.

(When added to Definition 2.1) the **above** rules in Definition 2.4 relate fluents with transition-attributes  $\{create, terminate, obfuscate\} \subseteq \mathcal{A}_t$  to the fluents with state-attributes  $\{created, terminated\} \subseteq \mathcal{A}_s$  in the post-transition state. For example,  $in((create, X), S)$  implies adding  $in((created, X), S + 1)$ . The correspondence suggested by the symmetry between names is formalised by the rules. However, there is a notable asymmetry: the state-attribute correspondent ‘obfuscated’ of *obfuscate* is not represented explicitly; it is implied by the absence of *created* and *terminated*, instead. The UPPER rules capture a form of three-value logic. The conditions on these rules recreate eFLINT’s existing, static prioritisation of different effects: termination is prioritised over obfuscation, and creation over obfuscation and termination.

The MIDDLE rules in Definition 2.4 attribute *holds* and *enabled* to instances, based on whether the instances are created, terminated, obfuscated, derived, or suppressed. The two rules attributing *holds* to instances show the distinct roles of creation and termination; in summary, instance  $I$  holds if it is either derived (and not terminated or suppressed) or created, thus abstracting away the details on how the truth of  $I$  was established. The *enabled* instances hold while not *suppressed*. Suppression lets eFLINT users express conditions that must always be true for an instance to be enabled, regardless of its other attributes. In practice, this lets (the makers of) eFLINT specifications (which use *suppress*) constrain the enabled actions, despite their lack of control over (the makers of) eFLINT scenarios (which use *create*).

Finally, the LOWER rules in Definition 2.4 define eFLINT’s dual notions of normative violation. Duty violations arise in states when enabled instances’ violation conditions are met. Action violations arise in transitions when disable instances are triggered. Note how the *enabled* attribute occurs in both rules; intuitively, this ensures that there is a finite search space of violation-free scenarios from any finite state; the finite enabled instances bound the permitted actions, and then it suffices to consider the next state’s enabled instances to check for violated duties. In eFLINT today, this kind of search problem is externalised, *e.g.*, to human users of the interpreter. But in Section 2.7, we internalise this kind of scenario-search problem via Clingo rules.

Examples 2.4 and 2.5 show state-internal Clingo rules that formalise clauses **Violated when Holds**(elapsed(deadline)) and **Derived from** (Foreach dataset: ... ) from of Example 2.1, respectively. The former is the result of the systematic translation of

an eFLINT specification clause in Section 2.6, while the latter is hand-crafted. Both demonstrate how Core eFLINT may be used as a language in its own right.

**Example 2.4** (must-notify violation condition as a Clingo rule).

```
in((violated,must_notify(U,C,A,D)),S) :- state(S) ;
    in((holds,elapsed(D)),S) ; in((enum,must_notify(U,C,A,D)),S}).
```

**Example 2.5** (default administrator control as Clingo rules).

```
in((derived,non_admin_control(dataset(D))),S) :-
    in((holds,controls(U,dataset(D))),S) ; not U = user("Admin") .

in((derived,controls(user("Admin"),dataset(D))),S) :-
    in((holds,dataset(D)),S) ; not in((holds,non_admin_control(dataset(D))),S) .
```

## 2.6 Specification Translation to Core eFLINT

This section defines the eFLINT specification language: its abstract syntax is a formal language  $\mathcal{S}$ , and its dynamic semantics is given by a translation from  $\mathcal{S}$  to Clingo.

### 2.6.1 Abstract Syntax

After Notation 2.2 introduces some basic notation, Figure 2.3 defines an abstract syntax for eFLINT specifications as its own formal language. It adheres closely to that which was originally defined in [vBLvDvE20] and then generalised in [vBKB<sup>+</sup>21]. Our minor deviations improve consistency and simplicity.

Each eFLINT *specification*  $s : \mathcal{S}$  consists of *type-definitions*: mappings from user-defined (*record or structure*) types  $t : \mathcal{T}$  to pairs of the form  $\langle V, C \rangle$ , respectively defining the *fields* and *clauses* of  $t$ . This definition enables the construction of  $t$ -type instances, whose structure is fixed by  $V = [v_1, v_2, \dots, v_n]$ ;  $t$  acts as a constructor for  $t$ -type instances from smaller instances.

Ultimately, Section 2.6.4 defines the dynamic semantics of each specification in terms of its clauses. Intuitively, each clause of each type  $t$  is evaluated in the context of an arbitrary *current* state  $S$  to give new attributes to  $t$ -type instances in  $S$ . Each clause fixes the attribute in question, but the instances arise dynamically, from the evaluation of instance expressions in  $\mathcal{E}$ . In the simplest case, the instance expressions *are* concrete instances (as  $\mathcal{I} \subset \mathcal{E}$ ). For example, *derive(struct(instant,[int(9)])*) gives the attribute *derived* to precisely the instance *instant(9)* in each state. The *for* construct generalises these concrete expressions to *lists* of instances, by quantifying over instances (already) with a given attribute, and binding these to variables in a new local scope. The *where* construct filters these lists. For example, *for(enabled,p,where(p,check(dutyViol,p)))* enumerates the current duty violation instances with the type of instance pattern  $p$ .

**Notation 2.2** (Common Types and Type-Combinators).

- $\mathbb{Z} \triangleq \{\dots, -2, -1, 0, 1, 2, \dots\}$  is the integer type.
- $\mathbb{S}$  is the type of string literals like "Well, howdy do?".
- $A \rightarrow B$  is the type of functions from  $A$  to  $B$ .
- $A \times B \triangleq \{\langle a, b \rangle \mid \forall a:A, b:B\}$  is the product of any gives types  $A$  and  $B$ .
- We use *set*, *map*, and *list* as the usual types of arbitrarily large but finite collections. For example,  $[x, y]$  is a list with two elements. We coerce  $\text{map}(A, B)$  to  $A \rightarrow B \cup \{\star\}$ , where  $\star \notin B$  marks each missing mapping, and to or from  $\text{list}(A \times B)$ , and we style these pairs  $\langle a, b \rangle$  suggestively as  $a \mapsto b$ . We coerce  $\text{set}(A)$  to and from  $\text{list}(A)$ .

$t, t', \dots : \mathcal{T} \triangleq \{\mathbb{Z}, \mathbb{S}, \dots(\text{more identifiers})\}$	(type ID)
$v, v', \dots : \mathcal{V} \triangleq \mathcal{T} \times \mathbb{S}$	(variable)
$g, g', \dots : \mathcal{G} \triangleq \text{sum} \mid \text{count} \mid \text{min} \mid \text{max}$	(aggregator)
$e, e', \dots : \mathcal{E} := \text{struct}(t, x : \text{list}(\mathcal{E})) \mid \text{int}(x : \mathbb{Z}) \mid \text{str}(x : \mathbb{S})$ $\mid \text{proj}(e, v) \mid \text{agg}(g, l) \mid \text{var}(v)$	(inst. <u>expr.</u> )
$p, p', \dots : \mathcal{P} \subseteq \mathcal{E}$ (just <i>struct</i> , <i>int</i> , <i>str</i> , <i>var</i> )	(inst. <u>patt.</u> )
$i, i', \dots : \mathcal{I} \subseteq \mathcal{P}$ (just <i>struct</i> , <i>int</i> , <i>str</i> )	(instance)
$l, l', \dots : \mathcal{L} := \text{for}(p, a, l) \mid \text{let}(p, e, l) \mid \text{where}(l, b) \mid e$	(inst. <u>list</u> )
$b, b', \dots : \mathcal{B} := \text{true} \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b$ $\mid e_1 < e_2 \mid e_1 = e_2 \mid \text{check}(a, e)$	(bool <u>expr.</u> )
$c, c', \dots : \mathcal{C} := \text{derive}(l) \mid \text{affect}(x : \mathcal{A}_t, l) \mid \text{actType}$ $\mid \text{infinite} \mid \text{filter}(a, b) \mid \text{finite}(x : \text{list}(\mathcal{I}))$	( <u>clause</u> )
$s, s', \dots : \mathcal{S} \triangleq \text{map}(\mathcal{T}, \text{list}(\mathcal{V}) \times \text{set}(\mathcal{C}))$	( <u>spec.</u> )
-----	
$\text{exists}(p, b) : \mathcal{B} \triangleq \text{int}(0) < \text{agg}(\text{count}, \text{for}(\text{enum}, p, \text{where}(p, b)))$	
$\text{forall}(p, b) : \mathcal{B} \triangleq \neg \text{exists}(p, \neg b)$	

Figure 2.3: Abstract syntax of eFLINT specifications  $\mathcal{S}$ , where the attributes  $a, a_1, a', \dots : \mathcal{A}$  are defined in Definition 2.3.

**Definition 2.5** (field pattern of type  $t$  in specification  $s$ ).

$$\text{fieldPatt}(t, s) \triangleq \text{struct}(t, \text{fields}) \text{ where } \langle \text{fields}, x \rangle \triangleq s(t)$$

$\text{type}(\text{agg}(g, l)) \mid$	$\text{type}(\text{int}(x)) \triangleq \mathbb{Z}$	$\text{type}(\text{str}(x)) \triangleq \mathbb{S}$
$\text{type}(\langle t, x \rangle : \mathcal{V}) \mid$	$\text{type}(\text{struct}(t, x)) \triangleq t$	$\text{type}(\text{proj}(e, v)) \triangleq \text{type}(v)$
$\text{type}(\text{where}(l, b)) \mid$	$\text{type}(\text{for}(p, a, l)) \triangleq \text{type}(l)$	

Figure 2.4: The typing function for instance (list) expressions.

Variables are the simplest patterns, binding quantified instances in their entirety. Other patterns quantify instances and bind their sub-instances to many variables. For example, if `for(struct(controls, [var(<agent, "x">, <dataset, "y">)]), enabled, l)` quantifies enabled `controls`-type instances, but binds its controller and dataset to variables named `x` and `y`, respectively. Like `for`, `let` binds instances to variables via patterns, but without any quantification.

## 2.6.2 Instance Types and Static Semantics

Only when type  $t$  has defined **fields**  $F : list(V)$  can any *instance*  $i : \mathcal{I}$  of type  $t$  be constructed.  $F$  determines the structure of  $t$ -type instances. Instances are constructed with `struct` and de-constructed (projected to a given field) with `proj`, and only when the instances are (de)constructed with the specified fields; e.g., `struct(t, [])` is well-typed iff  $t$  has fields `[]`. We require each types' fields to be distinctly identified (i.e., fields are map-like) such that each projection is unambiguous. Figure 2.4 defines the (static) typing function, such that evaluating each instance expression  $e : \mathcal{E}$  yields an instance  $i : \mathcal{I}$  with  $type(e) = type(i)$ . Example 2.6 demonstrates by specifying all types and fields (but omitting the clauses) of the running example.

**Example 2.6** (specifying fields (not clauses) of Example 2.1). For legibility, we use  $x$  to abbreviate each variable  $\langle x, "x" \rangle$ .

```
[
    instant ↦ ⟨[ℤ], []⟩,      user ↦ ⟨[S], []⟩,      dataset ↦ ⟨[S], []⟩,
    elapsed ↦ ⟨[instant],    [],
    controls ↦ ⟨[⟨user, "controller">, dataset], [],
    access ↦ ⟨[user, dataset, instant], [],
    must_notify ↦ ⟨[user, ⟨user, "controller">, access, ⟨instant, "deadline">]⟩, [],
    notify ↦ ⟨[user, ⟨user, "controller">, must_notify], [], ]
```

The *primitive types* integers ( $\mathbb{Z}$ ) and strings ( $\mathbb{S}$ ) are implicitly defined. Uniquely, their instances cannot be (de)constructed. Instead, `int` and `str` introduce their instances as literals in the program text. Integers also emerge dynamically, from inbuilt operators  $\{+, -, \times, \div, \%\}$ . We treat these as ordinary constants in the abstract syntax, but their special status is reflected by their infix notation in the concrete syntax, and by their reduction under evaluation. For example, `users` expresses `struct(+, int(2), int(3))` concretely as `2 + 3`, which is then reduced to `int(5)` under evaluation.

Finally, each eFLINT specification begins with the prelude `Fact int Identified by Int. Fact string Identified by String. Fact actor Identified by String..` Users consider `int`, `actor`, and `string` to be inbuilt, but they have no special properties.

The static semantics of eFLINT recognises specifications which only use types and variables as one might expect: 1. variables are bound to instances with matching types, 2. per  $struct(t, x)$ ,  $x$  binds each field defined for  $t$  once with none left unbound, 3. when any  $i$  is projected to  $v$  then  $x$  has a field  $v$ , and 4. accessed variables occur exactly once in patterns of an enclosing *let*- or *for*-expression.

### 2.6.3 Restricted Normal Form

Our translation accepts specifications in *restricted normal form*  $S' \subset S$ , which we characterise as the well-typed specifications that satisfy Properties 2.1 to 2.5.

**Property 2.1** (CNF Booleans). Boolean expressions are in conjunctive normal form.

Property 2.1 ensures that there is no  $\{\neg, \vee, \wedge\}$  inside any  $\neg e$ , and no  $\{\vee, \wedge\}$  inside any  $e_1 \vee e_2$ . This works around a syntactic limitation of Clingo: antecedents can express  $e_1 \vee e_2$  as  $e_1$ : **not**  $e_2$ , but such terms cannot be nested. Fortunately, we can apply standard algorithms to normalise our Boolean terms to satisfy Property 2.1.

**Property 2.2** (structured instances). In each  $for(e, a, l)$  and  $check(a, e)$ ,  $e \neq var(\dots)$ .

Ultimately, Property 2.2 ensures that eFLINT’s instance types are preserved in Clingo: each term matching  $in((A, e), S)$  necessarily quantifies instances of  $type(e)$ . Where  $type(e)$  is primitive, the expression is trivial, as primitive instances have no attributes. In the more difficult case, where  $e$  is some  $var(v)$ , we can statically *de-structure* the variable by replacing it with  $fieldPatt(type(v), s)$  (Definition 2.5) and renaming the new variables to avoid name collisions, if necessary.

**Property 2.3** (projection-freedom). No subterm matches  $proj(e, v)$ .

Property 2.3 reconciles a fundamental difference between how eFLINT and Clingo *destructure* terms. Clingo uses matching and unification to bind subterms to variables. In contrast, eFLINT uses *proj* to project structures to given fields, for which Clingo has no analogue. We reconcile this difference by applying eFLINT-style projection to each  $proj(e, v)$ , statically. Because the specification is well-typed,  $type(e)$  is not primitive, so  $e$  is either a variable or a structure. In the first case, where  $e = var(v)$ , we first ‘explode’ the  $var(v)$ , replacing it with  $fieldPatt(type(v), s)$  (Definition 2.5), renaming variables as needed. In both cases we get  $proj(struct(t, [e_1, e_2, e_3, \dots]), v)$ , which we rewrite to  $e_k$ , where  $v$  is the  $k^{\text{th}}$  field of  $t$ .

**Property 2.4** (no nested aggregators). In each  $agg(g, l)$ , term  $l$  contains no  $\{agg, \vee\}$ .

Fortunately, Clingo and eFLINT have corresponding syntax and semantics for aggregators. But unfortunately, only eFLINT permits aggregators to be arbitrarily nested. Property 2.4 works within this limitation. Any specification can be transformed such that it satisfies Property 2.4 by ‘extracting’ each nested aggregator  $agg(g, l)$  via *reification*: it is replaced by  $struct(t, \langle var(v) \rangle)$  where  $v$  is a fresh variable

of the fresh type  $t$ , defined with fields  $[\mathbb{Z}]$ ; intuitively,  $t$  collects the elements to be aggregated. In Example 2.5, `non_admin_control` demonstrates how the eFLINT expression `(Exists user: ...)` in Example 2.1 can be reified as a new predicate `non_admin_control`. But, as this contains no nested aggregators, Example 2.1 already satisfies Property 2.4; this particular eFLINT expression does not need to be reified.

**Property 2.5** (antecedent aggregators). Each  $agg$  occurs in the  $l$  of some  $let(p, e, l)$  or in the  $b$  of some  $where(l, b)$ .

Property 2.5 works around Clingo’s limitations on aggregations in consequents. For example, `f(x) :- x = #count{ g(y) }` is valid Clingo, but `f(#count{ g(y) })` is not. It suffices to replace  $agg(g, l)$  with  $let(var(v), agg(g, l), var(v))$  for a fresh  $v$ , which achieves a similar result; the aggregation becomes an antecedent.

## 2.6.4 Dynamic Semantics of eFLINT Specifications

The dynamic semantics of a specification  $s$  is defined as a translation to state-internal Clingo rules from the *clauses*  $C \subseteq \mathcal{C}$  associated with each type definition  $t \mapsto \langle V, C \rangle$  in  $s$ . The translation for a clause  $c \in C$  associated with  $t$  in  $s$  is denoted  $\llbracket s, t, c \rrbracket$ , where function  $\llbracket \cdot \rrbracket$  is defined in Figure 2.5. A complete translation applies this function to each clause in a given specification, in any order. The result is well-formed if the specification is of the restricted normal form, *i.e.*, it is well-typed, and satisfies Properties 2.1 to 2.5. The translation benefits from the declarative nature of Clingo. Notably, reordering the rules or adding white space never changes their meaning.

Figure 2.5 translate the six kinds of clauses to Clingo; each ultimately gives new attributes to (new) instances. The cases of the  $\llbracket \cdot \rrbracket$  definition show how the specialised semantics of these clauses are reconciled. For example, the translation makes explicit the instances which are implicitly quantified by *affect* clauses: each effect arises from some triggered instance. Ultimately, the translation proceeds by inductively unfolding syntactic sub-structures. For example,  $[\cdot]^{\mathcal{L}}$  separates rule-consequents from -antecedents, and  $[\cdot]^{\mathcal{G}}$  maps eFLINT’s aggregator syntax to the corresponding Clingo.

Much of the complexity of the translation represents a reconciliation of the semantics of eFLINT and Clingo. For example, clauses in eFLINT are evaluated in the context of an implicit current state, but our translation makes it explicit as the Clingo variable `s`. Some other complexities arise from the idiosyncrasies of eFLINT and Clingo. For example, Clingo’s concrete syntax differs inside and outside of aggregator expressions such as `#sum { ... }`; for example, Clingo rules separate antecedents from consequents with `:-` outside, but with `:` inside.

Example 2.7 demonstrates our translation of part of the running example: a *filter* clause chosen because it is already in the restricted normal form (see Section 2.6.3). As with the Clingo rules in Example 2.5, these rules are state-internal, applying to

$$\begin{aligned}
\llbracket s, t, \text{derive}(l) \rrbracket &\triangleq [l, \text{derived}]^{\mathcal{L}} . \\
\llbracket s, t, \text{affect}(l, a) \rrbracket &\triangleq [\text{for}(p, \text{trigger}, l), a]^{\mathcal{L}} . \\
\llbracket s, t, \text{filter}(b, a) \rrbracket &\triangleq [\text{for}(p, \text{enum}, \text{where}(p, b)), a]^{\mathcal{L}} . \\
\llbracket s, t, \text{infinite} \rrbracket &\triangleq [\text{for}(p, \text{holds}, p), \text{enum}] . \\
\llbracket s, t, \text{actType} \rrbracket &\triangleq [\text{for}(p, \text{trigger}, p), \text{actTrigger}] . \\
\llbracket s, t, \text{finite}(i_1, i_2, \dots, i_n) \rrbracket &\triangleq [i_1, \text{enum}]^{\mathcal{L}} . [i_2, \text{enum}]^{\mathcal{L}} . \dots [i_n, \text{enum}]^{\mathcal{L}} . \\
&\text{where } p \triangleq \text{fieldPatt}(t, s)
\end{aligned}$$


---

$$\begin{aligned}
[\text{for}(p, a', l), a]^{\mathcal{L}} &\triangleq [\text{where}(l, \text{check}(a', p)), a]^{\mathcal{L}} \\
[\text{where}(l, b), a]^{\mathcal{L}} &\triangleq [l, a]^{\mathcal{L}} ; [b]^{\mathcal{B}} \\
[\text{let}(p, e, l), a]^{\mathcal{L}} &\triangleq [l, a]^{\mathcal{L}} ; [p]^{\mathcal{E}} = [e]^{\mathcal{E}} \\
[e, a]^{\mathcal{L}} &\triangleq \text{in}(\langle [a]^{\mathcal{A}}, [e]^{\mathcal{E}} \rangle, \mathbb{S}) \text{ :- state}(\mathbb{S}) \\
[\text{true}]^{\mathcal{B}} &\triangleq \text{\#true} \\
[\text{check}(a, e)]^{\mathcal{B}} &\triangleq \text{in}(\langle [a]^{\mathcal{A}}, [e]^{\mathcal{E}} \rangle, \mathbb{S}) \\
[b_1 \wedge b_2]^{\mathcal{B}} &\triangleq [b_1]^{\mathcal{B}} ; [b_2]^{\mathcal{B}} \\
[b_1 \vee b_2]^{\mathcal{B}} &\triangleq [b_1]^{\mathcal{B}} : \left[ \begin{array}{l} b \text{ if } \exists b, \neg \neg b = \neg b_2 \\ \neg b_2 \text{ otherwise} \end{array} \right]^{\mathcal{B}} \\
[\neg b]^{\mathcal{B}} &\triangleq \text{not } [b]^{\mathcal{B}} \\
[e_1 < e_2]^{\mathcal{B}} &\triangleq [e_1]^{\mathcal{E}} < [e_2]^{\mathcal{E}} \\
[e_1 = e_2]^{\mathcal{B}} &\triangleq [e_1]^{\mathcal{E}} = [e_2]^{\mathcal{E}} \\
[\text{struct}(t, [e_1, e_2, e_3, \dots, e_n])]^{\mathcal{E}} &\triangleq t \langle [e_1]^{\mathcal{E}}, [e_2]^{\mathcal{E}}, [e_3]^{\mathcal{E}}, \dots, [e_n]^{\mathcal{E}} \rangle \\
[\text{agg}(g, l)]^{\mathcal{E}} &\triangleq [g]^{\mathcal{G}} \{ r([l, \text{enum}]^{\mathcal{L}}) \} \\
&\text{where } r \text{ replaces each } ; \text{ with } , \text{ and } \text{ :- } \text{ with } : \\
[r(x)]^{\mathcal{E}} &\triangleq x \text{ for } r \in \{\text{var}, \text{int}, \text{str}\} \\
[\cdot]^{\mathcal{G}} : \mathcal{G} \rightarrow \mathbb{S} &\triangleq \{\text{count} \mapsto \text{\#count}, \text{sum} \mapsto \text{\#sum}, \dots\} \\
[\cdot]^{\mathcal{A}} : \mathcal{A} \rightarrow \mathbb{S} &\triangleq \{\text{trigger} \mapsto \text{trigger}, \text{create} \mapsto \text{create}, \dots\}
\end{aligned}$$

Figure 2.5: This defines functions  $\llbracket \cdot \rrbracket$  and  $[\cdot]$  functions.  $\llbracket s, t, c \rrbracket$  is the translation to Clingo of the clause  $c$  in the definition of type  $t$  in (restricted) specification  $s$ .

each state  $s$  individually, relying on underlying Core eFLINT and state trace semantics to relate fluents within and between successive states, respectively.

Example 2.1 shows the full translation of Appendix B; readers can scrutinise the eFLINT and Clingo side-by-side to better understand the details our translation.

**Example 2.7** (translating the running example violation condition to Clingo).

We represent the concrete eFLINT clause `Violated when Holds(elapsed(deadline))` in Example 2.1 on the `must_notify` duty-type as the abstract eFLINT clause `filter(b, violated)`, where  $b \triangleq \text{holds}(\text{struct}(\text{elapsed}, [\{\text{instant}, \text{"deadline"}\}]))$ . In natural language, this clause gives attribute `violated` to each (implicitly) quantified `must_notify`-type instance  $i$ , on the condition that the deadline of  $i$  has elapsed. In the following, let  $p \triangleq \text{fieldPatt}(\text{must\_notify}, s)$ , *i.e.*,  $p$  abbreviates the field pattern of type `must_notify` in  $s$ , the running example specification; it suffices to recall from Example 2.6 that `must_notify` has fields of types `[user, user, access, data]`.

$$\llbracket s, \text{must\_notify}, \text{filter}(b, \text{violated}) \rrbracket$$

where  $b \triangleq \text{holds}(\text{struct}(\text{elapsed}, [\{\text{instant}, \text{"deadline"}\}]))$

$$\Rightarrow [\text{for}(p, \text{enum}, \text{where}(p, b)), \text{violated}]^{\mathcal{L}} .$$

where  $p \triangleq \text{fieldPatt}(\text{must\_notify}, s)$

$$\Rightarrow [\text{where}(\text{where}(p, b), \text{check}(\text{enum}, p)), \text{violated}]^{\mathcal{L}} .$$

$$\Rightarrow [\text{where}(p, b), \text{violated}]^{\mathcal{L}} ; [\text{check}(\text{enum}, p)]^{\mathcal{B}} .$$

$$\Rightarrow [p, \text{violated}]^{\mathcal{L}} ; \text{in}((\text{holds}, [b]^{\mathcal{B}}), \mathcal{S}) ; \text{in}((\text{enum}, [p]^{\mathcal{E}}), \mathcal{S}) .$$

$$\Rightarrow [p, \text{violated}]^{\mathcal{L}} ; \text{in}((\text{holds}, \text{elapsed}(\mathcal{D})), \mathcal{S}) ; \text{in}((\text{enum}, [p]^{\mathcal{E}}), \mathcal{S}) .$$

$$\Rightarrow \text{in}((\text{violated}, [p]^{\mathcal{E}}), \mathcal{S}) \text{ :- state}(\mathcal{S}) ; \text{in}((\text{holds}, \text{elapsed}(\mathcal{D})), \mathcal{S}) ; \text{in}((\text{enum}, [p]^{\mathcal{E}}), \mathcal{S}) .$$

$$\Rightarrow \text{in}((\text{violated}, \text{must\_notify}(\mathcal{U}, \mathcal{C}, \mathcal{A}, \mathcal{D})), \mathcal{S}) \text{ :- state}(\mathcal{S}) ;$$

$$\text{in}((\text{holds}, \text{elapsed}(\mathcal{D})), \mathcal{S}) ; \text{in}((\text{enum}, \text{must\_notify}(\mathcal{U}, \mathcal{C}, \mathcal{A}, \mathcal{D})), \mathcal{S}) .$$

**Our Generalisations** Our version of the eFLINT specification language generalises the original in three ways. Firstly, our *let* binders are new. Secondly, we generalise binders in *for* (and *let*) from variables to patterns. Thirdly, our *for* quantifies over instances with any attribute and not just the *enumerables* (with *enum*). In principle, these generalisations enable new usages, but we only use them *within* the translation: for transformations described in Section 2.6.3; *e.g.*, our experiments use none of these generalisations. But Section 2.10 discusses our interest in exploring eFLINT variants.

## 2.7 Scenario Translation to Core eFLINT

This section completes the definition of eFLINT by defining the scenario language and translating it to Clingo. We distinguish the two scenario representations, reflecting the two use cases of our new interpreter. In either case, we assume we are given an eFLINT specification, which has already been translated into Clingo rules.

### 2.7.1 Use Case: Check if a Scenario is Compliant

The first use case is scenario-compliance (demonstrated in Example 2.1) in which the interpreter checks the compliance of a given scenario. Precisely, each scenario is in  $list(\mathcal{A}_t \times I)$ . Intuitively, each scenario labels each  $N^{\text{th}}$  transition with an *effect*: assigning a transition-attribute to a given instance. Example 2.8 shows the running example scenario before and after translation. Note that the translation is straightforward, and the resulting rules are state-internal.

**Example 2.8** (three encodings of the scenario in Example 2.1).

Concrete eFLINT	Abstract eFLINT	Clingo encoding
+dataset("X-Rays").	[⟨create, dataset("X-Rays")⟩,	in((create, dataset("X-Rays")), 1).
+controls(user("Amy"), dataset("X-Rays")).	⟨create, controls(user("Amy"), dataset("X-Rays"))⟩,	in((create, controls(user("Amy"), dataset("X-Rays"))), 2).
+user("Bob").	⟨create, user("Bob")⟩,	in((create, user("Bob")), 3).
+instant(9).	⟨create, instant(9)⟩,	in((create, instant(9)), 4).
access(user("Bob"), dataset("X-Rays")).	⟨trigger, access(user("Bob"), dataset("X-Rays"), 9)⟩]	in((trigger, access(user("Bob"), dataset("X-Rays"), 9)), 5).

In the dynamic semantics of a scenario, a concrete ‘current state’ is modified to its successor state in between the elements of a scenario. Recall that the dynamic semantics of a specification (Section 2.6) refers to an abstract current state  $s$  and assigns attributes to instances in  $s$ . The rules generated from an eFLINT specification – alongside the rules of Core eFLINT (Section 2.5) and state traces (Section 2.4) – thus produce attribute-assignments for all concrete states generated by the scenario according to the clauses of the specification. It is precisely these attribute-assignments that are interesting output to the eFLINT user. For example, the output of Example 2.1 enumerates the instances derived throughout the scenario, which includes the administrator’s control, derived by default, in `state(1)`.

### 2.7.2 Use Case: Search for Satisfactory Scenarios

In this novel usage, the user encodes a scenario-search problem: Clingo rules which define 1. the search space of possible scenario actions, *e.g.*, via Clingo’s *disjunctive-head* rules, and 2. the satisfaction criterion, *e.g.*, via Clingo’s constraint rules.

The search problems expressible via these rules enjoy the same freedoms and are subject to the same limitations as expressing search problems with Clingo in general. For example, the search space and search criterion do not need to be state-internal.

Example 2.9 shows the Clingo encoding of a search-problem that is of particular interest to eFLINT users and the normative domain in general. Precisely, Example 2.9 *model-checks* the specification against the proposition ‘taking only enabled actions violates no duties’, captured by the final rule in the negative, such that stable models encode counterexamples. Each solution details one offending counterexample scenario.

**Example 2.9** (find a scenario that is act-compliant but not duty-compliant).

```

----- the case-generic part -----
1 = { choose(X,S) : triggerable(X,S) } :- state(S).
in((trigger,X),S) :- choose(X,S).
:- 0 = { in((dutyViol,X),S) }.

----- the case-specific part -----
{ state(S) } :- S = 1..1000.
triggerable(X,S) :- state(S + 1) ; in((enabled,access(U,D,I)),S)).
triggerable(X,S) :- state(S + 1) ; in((enabled,notify(U,C,M)),S)).
in((create,controls(user("Amy"),dataset("X-Rays" )),1).
in((create,controls(user("Amy"),dataset("Cat Scans")),1).
in((create,controls(user("Bob"),dataset("Allergies")),1).
in((create,controls(user("Dan"),dataset("CT-Scans" )),1).

```

The case-specific part defines the search space: its depth is limited to 1000 states, and its breadth is defined by the *triggerable* (enabled *access-* or *notify-*type) instances, which result from the initially created *control*-instances. This input yields no stable model; the property holds! But the scenario is not interesting because no actions advance the *elapsed* time. Adding `in((create,elapsed(instant(S))),S) :- state(S)` conflates the passage of states with the passage of time; then counter-examples arise, *e.g.*, where Dan accesses the X-Rays, but violates the duty to inform Amy at instant 11.

## 2.8 Implementation Overview

The artefact accompanying this chapter at [MEvB25] includes the Haskell source code of our new eFLINT interpreter, which implements our new semantics.

**Input Language** Our new implementation is a fork of the existing eFLINT interpreter implementation, and we have kept its front-end unchanged: its parser, tokeniser, and de-sugarer. Consequently, the two interpreters have exactly the same concrete eFLINT input language, *e.g.*, as is shown in the running example (Example 2.1).

**Implementation Internals** Like the original, our new implementation accepts any eFLINT specification-scenario pair as input, interprets it, and outputs the logical

consequences and violations. Internally, our new implementation processes the input in the following stages:

1. The entry point is a Python script, parametrised with the path to the input(s). The script invokes our fork of the eFLINT interpreter implementation in Haskell.
  - a) The input eFLINT is parsed, tokenised, and de-sugared, using the existing interpreter front-end, resulting in the original implementation’s internal representation: scenarios and *unrestricted* eFLINT specifications.
  - b) A new function applies the transformations described in Section 2.6.3, mapping to our restricted normal form, encoded in a simple Haskell-embedding of our eFLINT abstract syntax that is shown in Figure 2.3.
  - c) The internal representation is translated into a combined Clingo ruleset  $R_1$ , using the translations that are detailed in Sections 2.6 and 2.7 for specifications and scenarios, respectively.
  - d)  $R_1$  is appended to another, fixed ruleset  $R_2$ : the Clingo rules encoding the semantics of state traces and Core eFLINT in Definitions 2.1 and 2.4 and presented in Sections 2.4 and 2.5, respectively.  $R_1 \cup R_2$  is returned.
2. The Python script invokes the Clingo reasoner with the ruleset  $R_1 \cup R_2$  as input in the ‘enumerate all (stable models)’ mode. The resulting Clingo output is a set of stable models, each consisting of a set of *true* Clingo terms.

**Output Language** The two implementations have the same output language: an enumeration of the fluents in each state and transition. Both implementations also have identical encodings of *instances* such as `controls("Amy", "X-Rays")`: each is a (nested) tuple of its arguments preceded by its type identifier, as shown throughout this chapter. But otherwise, the ordering of the fluents and their encoding of attribute information differs between the two implementations. The differences are ultimately reconciled by a utility script, which is included in our artefact, and which we use throughout our experiments to compare the two implementations’ outputs.

The original interpreter is an interactive read-evaluate-print-loop (REPL): users interleave the introduction next scenario steps with queries of the current state. Transition-fluents are extracted from the interpreter’s output in reaction to each next scenario step: it enumerates each newly created, terminated, and obfuscated instance is enumerated with a preceding (+), (-), and (~), respectively. Triggered instances have no prefix. In the case the effect is a triggered instance, the consequent effects are enumerated as a tree, *i.e.*, because triggers can cause new triggers. Also, triggered instances are annotated: whether they are also enabled (ENABLED) or not (DISABLED). State-fluents are extracted from the interpreter’s response to the query `:d` (‘show all

contents of the current [state]’), which enumerates the instances with attributes *holds* and *terminated* as instances suffixed by = **True** and = **False**, respectively. After each transition, each duty violation is enumerated with a preceding **violated duty!**:

Our new implementation relies on Clingo to enumerate all states and their fluents explicitly, in alphabetic order. Fluents are encoded as shown in Section 2.6.

Aside from the superficial differences in how attributes are expressed, there are only two significant considerations in reconciling the two interpreters’ outputs. Firstly, not all fluents are observable using the original interpreter. For example, the *suppressed* and *derived* only influence the original interpreter’s output indirectly; they are never enumerated and cannot be queried, but they affect which instances hold. Secondly, the output of the original interpreter uses *delta encoding* in reporting which instances *become* created and terminated in each new state. In contrast, Clingo explicitly enumerates each fluent in each state. To demonstrate the difference, consider the running example; the original interpreter (only) reports that `dataset("X-Rays")` is created after the 1<sup>st</sup> transition, but we can infer that it remains created in the 4<sup>th</sup> state, because in that state the response to the query `:d` includes `dataset("X-Rays") = True`.

## 2.9 Implementation Evaluation

We experimentally compare the behaviours of the two eFLINT interpreter implementations. In each experiment, each interpreter is given the same input eFLINT specification-scenario pair, using the existing concrete eFLINT language syntax.

**Scope: Scenario Search** We focus on the existing use case of *checking* given scenarios (see Section 2.7.1), because our search for scenarios is incomparable to in the existing interpreter, and because the performance of Clingo is extensively evaluated elsewhere [BBF23, HCP22]. Section 2.10 discusses planned future evaluations, for example, by comparing our scenario-search to other languages’ model-checkers.

**Reproducibility** The accompanying artefact includes our source code, experimental data, scripts, and a command-line tool for comparing behaviours [MEvB25]. Figure 2.6 exemplifies the output of this tool: it parses the output of each eFLINT interpreter, and compares the results side-by-side in the style of a Git ‘diff’.

**Test Suite** Our test suite contains many specification-scenario pairs. At the time of writing, there are 108 pairs (43 specifications and 108 scenarios), but we expect to continue adding tests. Most tests are small and hand-crafted to test individual facets of the semantics. For example, three distinct cases test the behaviour of 1. simultaneously creating many instances, 2. simultaneously creating and terminating one instance, and 3. simultaneously suppressing and deriving one instance.

```

Running diff test ./tests/clingo/diff_4_exists_forall.diff0000.eflint...
1 +-eFLINT-----+Clingo-----+
  | (holds, meta("")) | (holds, meta("")) |
  | (holds, no_dubs("")) | (holds, no_dubs("")) |
2 +-eFLINT-----+Clingo-----+
< | (holds, meta("")) | |
  | (holds, no_dubs("")) | (holds, no_dubs("")) |
  | (holds, some("")) | (holds, some("")) |
  | (holds, x(2)) | (holds, x(2)) |
3 +-eFLINT-----+Clingo-----+
< | (holds, meta("")) | |
  | (holds, some("")) | (holds, some("")) |
  | (holds, x(2)) | (holds, x(2)) |
  | (holds, x(3)) | (holds, x(3)) |
4 +-eFLINT-----+Clingo-----+
< | (holds, meta("")) | |
  | (holds, no_dubs("")) | (holds, no_dubs("")) |
  | (holds, some("")) | (holds, some("")) |
  | (holds, x(3)) | (holds, x(3)) |
5 +-eFLINT-----+Clingo-----+
  | (holds, meta("")) | (holds, meta("")) |
  | (holds, no_dubs("")) | (holds, no_dubs("")) |
+-----+
> Test OK
1 test(s) pass, 0 test(s) fail, 0 performance test(s)

```

Figure 2.6: A screenshot of our prototype command-line visualiser tool, comparing the outputs of our new eFLINT implementation (right) against the original (left), for the same specification-scenario input pair. This shows the result of some test input (in the supplement) where three fluents are inferred by only the original interpreter. This test passes because the difference is expected in this case.

Our test suite also includes two larger tests, which evaluate many facets of the semantics in combination. Firstly, we include the running example (Example 2.1) in its entirety; as explained in Section 2.1, the interpreters have identical behaviour only *after* the type `user` is renamed to `agent`, whereafter the interpreters implement the same reasoning by default, reaching the same states with the same fluents. Secondly, the test suite includes the *Algemene Afsprakenstelling*<sup>3</sup>, a formalisation of the data sharing agreement of the Dutch Metropolitan Innovations Ecosystem (DMI) consortium, which is (to our knowledge) the most complex and realistic eFLINT specification at the time of writing. For example, this document stipulates how the consortium’s yearly ‘commons’ budget may be spent by its members.

The experimental results to show only an illustrative subset of the entire test suite. Please see the artefact for the entire test suite.

## 2.9.1 Correctness Evaluation

We characterise the correctness of our new eFLINT semantics as matching input/output behaviours of the two implementations, where expected.

<sup>3</sup>The document is available at [https://dmi-ecosysteem.nl/wp-content/uploads/bb\\_documents/2023/10/2023.06.01-DMI-Afsprakenstelling-v1.pdf](https://dmi-ecosysteem.nl/wp-content/uploads/bb_documents/2023/10/2023.06.01-DMI-Afsprakenstelling-v1.pdf), our formalisation process is documented at <https://definitives.dmi-ecosysteem.nl>. Snapshots of both of these documents are stored alongside our eFLINT formalisation in the supplementary artefact.

**Differences in Reasoning by Default** The implementations are expected to disagree only in cases of *reasoning by default* where conclusions are conditioned on the implicit or explicit *absence* of fluents, namely, the evaluation of *where(l, check(...))* and *agg(...)*, change when fluents are removed. The `Not(Exists...)` expression in Example 2.1 is such a case. From the user perspective, the existing interpreter can ‘overlook’ conditions of this form. This emerges from the reasoning algorithm it implements; in summary, it interleavedly evaluates expressions and populates a collection of fluents with the valuation *true*, but it does not keep track of which fluents were previously checked for *falsity*, *i.e.*, absence from the collection, and so, additions can overturn prior checks. Consequently, the order in which clauses are evaluated and instances are quantified affects the output. The trouble is that this ordering is sensitive to incidental details of the specification; recall from Section 2.1 that this includes sensitivity to type identifiers; in that case, renaming `user` to `agent` changed the result. In fact, the result is also changed if the first scenario step (`+user("Amy")`) is omitted; unexpectedly, it matters whether `user("Amy")` holds because it was created in the 1<sup>st</sup> transition or because it is derived in the 4<sup>th</sup> state from `controls(user("Amy"), dataset("X-Rays"))`.

**Different Canonical Models** First, we focus on cases where, given the same input, the interpreters both return a single (‘canonical’) model, but the models are different.

First, Experiment 2.1 shows the case of a very small input, where only two rules are applied in total. Here, the original interpreter’s behaviour is sensitive to a superficial syntactic features of the specification. Consequently, only minor transformations to these inputs are required to force the original interpreter to apply the specified derivation rules in the right order, and thus, correctly reason by default.

**Experiment 2.1** (the original interpreter is sensitive to type naming). For the following two specifications, both versions of the eFLINT interpreter agree that only `y("")` holds. Indeed, these specifications have the same stable model, constructed by applying the second derivation clause, after which the first clause is not applicable.

```
Fact x Identified by String Derived from x("") Where Not(Holds(y("")))
```

```
Fact y Identified by String Derived from y("")
```

---

```
Fact z Identified by String Derived from z("") Where Not(Exists y: True)
```

```
Fact y Identified by String Derived from y("")
```

Our new interpreter reaches the same result given the following, third version of the specification. In contrast, the order in which the original interpreter evaluates this third version of the specification is (mysteriously) disturbed, changing its meaning. It lets `x("")` hold, ‘forgets’ the prior condition on `y("")`, and finally lets `y("")` also hold.

```
Fact x Identified by String Derived from x("") Where Not(Exists y: True)
```

```
Fact y Identified by String Derived from y("")
```

Experiment 2.2 shows another case where the original interpreter behaves unexpectedly. The problem is that, unlike Experiment 2.1, a more significant reformulation of the specification is required to force the derivation rules to be ordered correctly. In general, the original interpreter tends to interleave derivations of different instance types, but here, primes can be derived ‘spuriously’ if the clause deriving primes is applied before rules deriving integers have already been applied.

**Experiment 2.2** (where re-ordering the rules is more difficult). The following specification formalises a dependency between derivations. For each  $2 \leq I \leq P$ , the rule deriving `prime(P)` depends on whether `int(I)` is derived. These dependencies are acyclic, so the rules can be ordered such that the conditions of each is preserved by later rules. For any scenario `+int(N)`, our interpreter derives the expected primes: `prime(M)` holds iff  $M$  is prime and  $M \leq N$ . But the original interpreter lets each integer be prime. This is a consequence of it applying rules in the wrong order; it repeatedly ‘overlooks’ smaller integers. This problem arises no matter how these type-definitions and clauses are re-ordered, or how identifiers are renamed.

```
Extend Fact int Derived from (Foreach int: int(int - 1) Where 2 < int)
Fact prime Identified by Int Derived from (Foreach int: prime(int)
  Where Not(Exists int1, int2: 1 Where int1 * int2 == int))
```

The following specification postpones the derivation of prime instances (to state 2), such that they are derived only after the holding integers are finalised (in state 1). Now both interpreters derive the expected primes for any scenario `addleq(N)`.

```
Fact prime Identified by Int Derived from (Foreach int: prime(int)
  Where Not(Exists int1, int2: 1 Where int1 * int2 == int))
Event addleq Related to int Creates int Syncs with addleq(int - 1) Where 2 < int
```

We posit that the new interpreter has the superior behaviour in these cases, because the alternative behaviour of the original interpreter has two major drawbacks in practice. Firstly, users experience specifications as *brittle*, *i.e.*, small changes to the input can have large, unexpected consequences on the output. Secondly, users experience conditions in clauses as *unreliable*, *i.e.*, under surprising circumstances, the meaning of a specification is unaffected by the inclusion of some of its conditions. For example, eFLINT programmers cannot rely on the original interpreter to enforce the mutual exclusive applicability of two clauses `Derive isReady(agent) Where Holds(ready(agent))` and `Derive unReady(agent) Where Not(Holds(ready(agent)))`, despite it seeming obvious.

**Differences in the Absence of Canonical Models** The original interpreter guarantees a *canonical (logical) model* for every input: it assigns one Boolean truth value to every conceivable query. Unfortunately, this behaviour can give users a false sense of security if they expect their specifications and scenarios to encode logical relationships. The problem arises in inputs expressing logical relationships which

have no sensible canonical meanings. We distinguish two cases: 1. the input has no logical interpretation, because it is contradictory, and 2. the input has several distinct and equally sensible logical interpretations. The stable model handles these cases by attributing a non-singleton set of stable models to the input, enumerating its distinct logical interpretations. Experiments 2.3 and 2.4 demonstrate. In the former, there is no stable model, so queries have no sensible answer. In the latter, there are several stable models, so queries only have sensible answers *per model*.

**Experiment 2.3** (no stable model). Given the following input, our interpreter gives no stable model, which is distinct from one empty stable model. The original interpreter lets  $p(\text{""})$  hold, because it applies the only derivation rule, overlooking the fact that this immediately overturns the condition  $\text{Not}(\text{Holds}(p(\text{""})))$ . Neither model  $\{\}$  nor  $\{ p(\text{""}) \}$  satisfy the logical formula  $p \leftrightarrow \neg p$  asserted by this specification.

```
Fact p Identified by String Derived from p(" ") Where Not(Holds(p(" ")))
```

**Experiment 2.4** (several stable models). Given the following input, our interpreter gives two stable models: in one,  $f(\text{"A"})$  holds but  $f(\text{"B"})$  does not, and *vice versa* in the other. The original interpreter agrees with just the second stable model.

```
Fact f Identified by String Derived from f("A") Where Not(Holds(f("B")))
                Derived from f("B") Where Not(Holds(f("A")))
```

We posit that users of our interpreter are better informed in these cases. In Experiment 2.3, its output makes the unsatisfiability of the specification explicit, rather than silently violating the logical formula encoded in the derivation clause. In Experiment 2.4, its output recognises logical ‘nondeterminism’, enumerating the consistent interpretations of the specification, rather than arbitrarily picking one.

**Identical Behaviour Otherwise** The two interpreters exhibit equivalent output behaviours precisely when expected. Their behaviours differ only in cases of reasoning by default; of the subset of tests shown in this chapter, these are Experiments 2.1, 2.3, and 2.4. We conclude that we have correctly redefined and reimplemented the original eFLINT semantics, except for the aforementioned intentional differences.

## 2.9.2 Performance Evaluation

Here, we report on the comparison between the two interpreters’ performance in evaluating the same inputs and producing the same output. Note that we run both interpreters in single-threaded mode; the original eFLINT interpreter has no data-parallelism features at present, and our cursory experiments found that the Clingo solver achieves no significant speedup from multithreading for inputs with single stable models, which is the case for all our test cases in this section.

**Hypothesis** *A priori*, we hypothesised that Clingo’s native implementation and extensive performance engineering would speed up the new interpreter’s reasoning with the ‘semantic’ complexity of the input: the number of applied rules and the size of the resulting model. For example, we expected Clingo to significantly outperform eFLINT in the case of inputs with many complex reasoning steps in each state, *e.g.*, where many derivation rules apply. However, we expected this speedup to be undercut in cases where the original interpreter exploits specialised characteristics of eFLINT, *i.e.*, in these cases, our Clingo approach would proceed naïvely. Firstly, because only the new interpreter must translate from eFLINT to Clingo, we expected overhead in cases where the syntactic input is proportionately large and complex, but its stable models are small and simple. In the extreme case, the translation predominates, and Clingo infers no fluents at all. Secondly, because the original implementation is specialised to a characteristic of the eFLINT semantics, we expected runs with long scenarios to slow down; the original interpreter fundamentally reasons one state and transition at a time, while our Clingo encoding may be subject to slowdown as Clingo wastes time attempting to apply rules to fluents in unrelated states.

**Test Setup** Each measurement is the median of 10 runs on a machine with the following specifications: { processor: Intel core i9 9th generation (16 hardware threads), memory: 32GB RAM DDR4, storage: NVMe hard drive }.

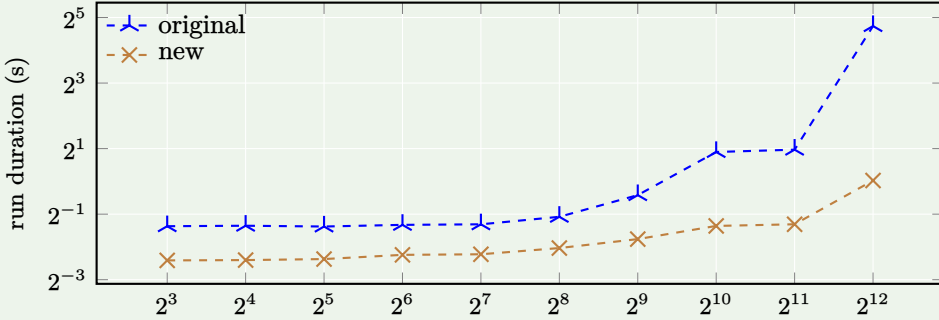
**Where the New Interpreter is Always Faster** First, Experiment 2.5 evaluates the performance of the interpreters for a pathological input, where inference grows an ever-increasing collection of facts and concrete rules with satisfied conditions, but where exactly one rule is applicable at a time, inferring  $\times(N - 1)$  from  $\times(N)$  and finishing with  $\times(\emptyset)$ . We observe that both interpreters run longer, superlinearly, with the length of the inference chain, presumably as a consequence of reasoning about ever-larger collections of inapplicable rules. But the new interpreter starts faster, and then slows down more gradually. The net result is that, compared to the original interpreter, the new interpreter begins with a  $2\times$  speedup, which grows to  $25\times$  for the most complex inputs. We have no explanation for  $N = 2^{11}$  representing a significant outlier, but we note that it was reliably reproduced by both interpreters.

Experiment 2.6 shows a case where achieved approximately  $2\times$  speedup in each scenario. Apparently, some task that is common to both interpreters predominates the cost of selecting and testing rules, which the two interpreters do differently. Because the output size of these inputs is consistently small, we expect the cost of dividing integers ( $/$ ) to predominate. However, further investigation is needed to be certain.

Experiment 2.7 shows the case of a scenario that was crafted to emphasise the expected strengths of Clingo: rules require quantifying many combinations of terms, performing many valuations and comparisons between terms. The cost of reasoning

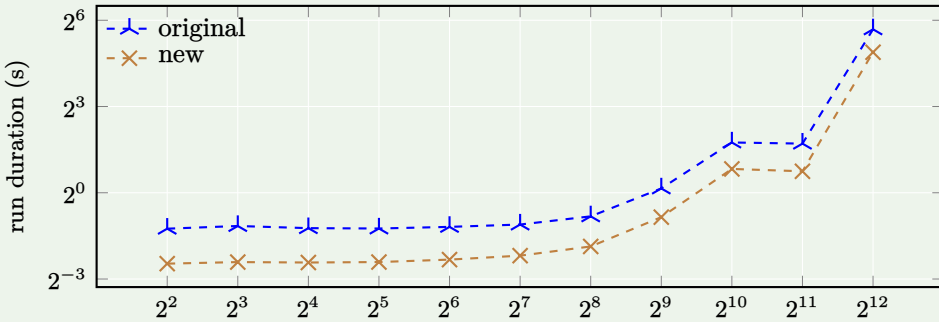
**Experiment 2.5** (deriving each preceding natural number).

Specification: `Fact x Identified by int`  
`Derived from (Foreach x: x(x.int - 1) Where 0 < x.int)`  
 Scenario: `(+x(N).)` for  $N \in \{8, 16, 32, \dots, 4096\}$



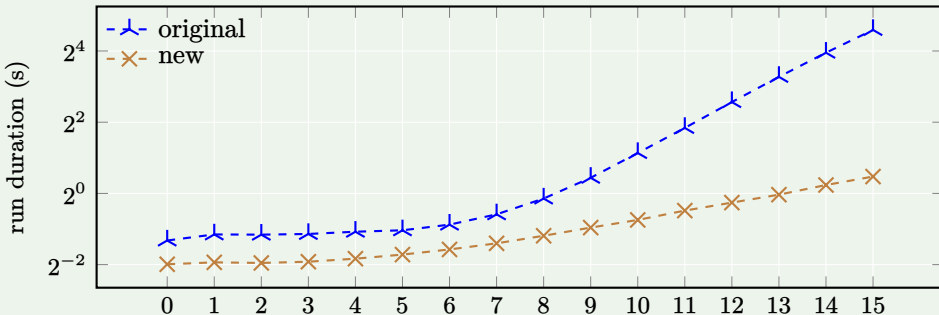
**Experiment 2.6** (deriving midpoints of existing integer pairs).

Specification: `Fact x Identified by Int`  
`Derived from (Foreach x1, x2: x((x1 + x2)/2))`  
 Scenario: `(+x(0). +x(N).)` for  $N \in \{0\} \cup \{4, 8, 16, \dots, 4096\}$



**Experiment 2.7** (complex checks conditioning derivation of integer triples).

Specification: `Fact x Identified by Int`  
`Derived from (Foreach y: y.x1), (Foreach y: y.x2)`  
`Derived from (Foreach y: y.x3), (Foreach x: x(x - 1) Where 0 < x)`  
`Fact y Identified by x1 * x2 * x3`  
`Derived from (Foreach x: y(x,x,x)), (Foreach x1, x2, x3:`  
`y(x1,x2,x3) Where (x1 == x2 || x2 != x3) && (Exists y: x2 < y.x1))`  
 Scenario: `(+x(N).)` for  $N \in \{0, 1, 2, \dots, 15\}$



about these scenarios scale more sharply than those of Experiment 2.5, before, but the speedups of the new interpreter over the original are similar: the simpler scenarios achieve  $2\times$  speedup, and the largest scales up to  $17\times$  speedup.

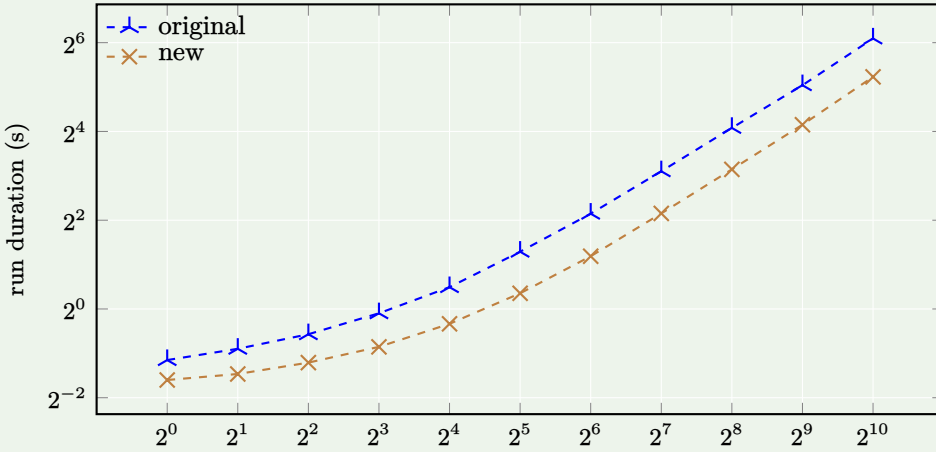
Experiment 2.8 shows the case where (the same) reasoning is spread over many states. We expected the original reasoner to excel in this situation, because it is implemented with eFLINT’s scenario-testing usage in mind, so its implementation exploits a fundamental ‘causality’ property of its semantics: all fluents in the  $N^{\text{th}}$  depend only on fluents in the  $N^{\text{th}}$  state and the  $N - 1^{\text{st}}$  transition. It implements the intuition: fluents of each state and transition are populated interleavedly. Precisely, we have formalised this property of eFLINT as all Clingo rules being state internal (Definition 2.2), except the state trace semantic Rule (INERTIA), which infers fluents each state from the fluents of its predecessor. In contrast, our new interpreter is naïve in this respect, because all states and transitions are encoded at once, and Clingo cannot be relied upon to recognise and exploit the subtle causality property. As expected, our interpreter consequently scales superlinearly with the length of the scenario, because the rules reasoning in each state are evaluated alongside unrelated rules, which nevertheless consume resources. For example, they certainly consume memory, and they possibly consume time (besides cache misses), as Clingo fruitlessly applies tries rules in the wrong order. Indeed, Experiment 2.8 shows that, after peaking at  $N = 2^6$  with 94% speedup, the new interpreter’s begins to lose speedup. However, surprisingly, the decrease is far slower than we expected. It was impractical to test an input size large enough to test whether the runtime curves ever cross. In our largest test case, with  $N = 2^{10}$ , the new interpreter achieved a speedup of 82%.

**Where the Old Interpreter is Circumstantially Faster** Experiment 2.9 compares the runtime performance of the eFLINT specification for the computation of primes, where the two interpreters agree on the result: the 2<sup>nd</sup> specification shown in Experiment 2.2. In this case, the original interpreter consistently outperforms our new interpreter for  $N \in \{50, 75, 100, \dots, 225\}$ , but for neither smaller nor larger  $N$ . At the most extreme, the original interpreter achieves 96% speedup over the new one! From Experiment 2.8, we conclude that the original interpreter’s sequential reasoning about the two states is not the cause for its speedup. We have a plausible explanation for this surprising result, but we leave its validation out of scope. We posit that the prior results that showed Clingo’s superior performance resulted from its successful application of appropriate optimisations, *e.g.*, heuristics which apply rules in an efficient order. But this specification encodes a problem which Clingo fails to optimise, so it falls back on a more naïve approach to evaluating rules. This is conceivable, because this problem encodes the *primality test* problem, which is famously hard; *e.g.*, in 2004, Agrawal, Kayal, and Saxena published their work

**Experiment 2.8** (the new interpreter excels unexpectedly at long scenarios).

Specification: Specification in Experiment 2.7.

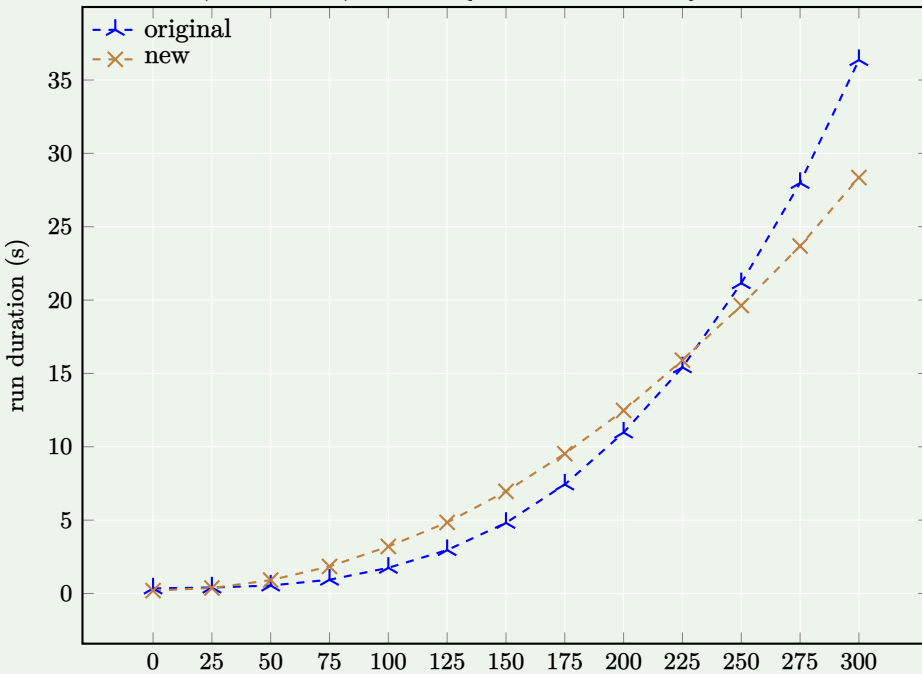
Scenario: `(+x(4). concatenated N times)` for  $N \in \{0\} \cup \{2^0, 2^1, 2^2, \dots, 2^{10}\}$



**Experiment 2.9** (the old interpreter unexpectedly can excel at computing primes).

Specification: 2<sup>nd</sup> specification in Experiment 2.2.

Scenario: `(add_1eq(N).)` for  $N \in \{0, 25, 50, 75, \dots, 300\}$



on their P-time primarily test algorithm [AKS04], for which they won the coveted Gödel prize.<sup>4</sup> Without an advantage in ordering rules, the new interpreter is at a disadvantage, because our encoding imposes overhead in the form of terms it must output explicitly (*e.g.*, *derived* instances) which the original interpreter handles only internally. Whatever the reason, the original interpreter is outperformed by the new interpreter when  $N > 255$ , where Clingo’s optimisations evidently predominate.

**Optimising the Core eFLINT Rules** Experiment 2.10 compares the relative performance of the new interpreter before and after the Core eFLINT rules are hand-optimised to exploit unique properties of the input specification. These optimisations

<sup>4</sup>I expect that most readers will not need to be convinced that primality testing would be hard to find and solve efficiently in arbitrary Clingo programs. But I make this point for completeness.

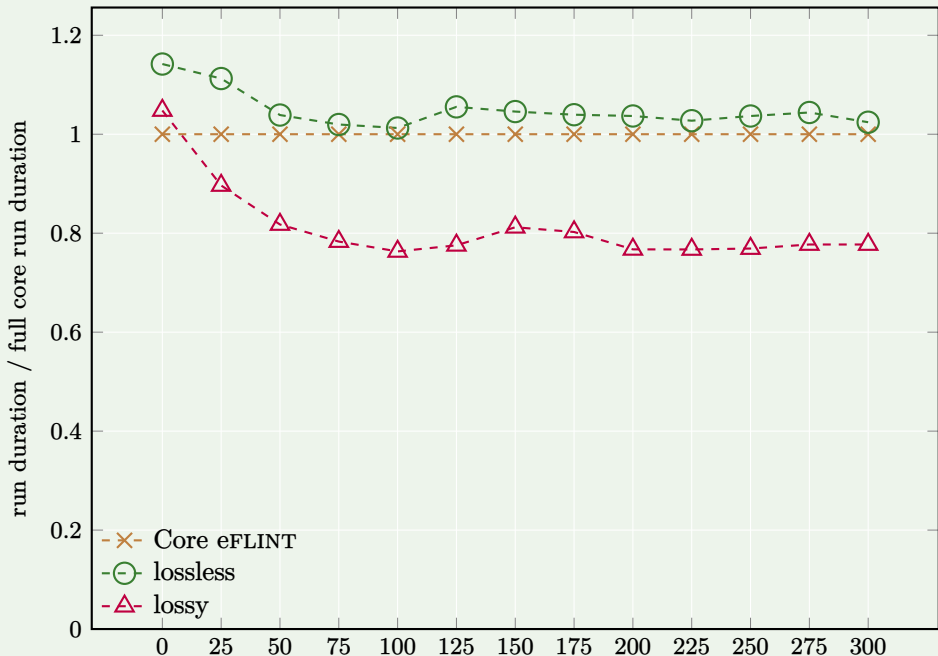
**Experiment 2.10** (optimising Core eFLINT). We let the new interpreter repeat Experiment 2.9, before and after the Core eFLINT rules in Definition 2.4 are replaced with the following rule sets. The ‘lossless’ rules produce identical output *for this input*. The ‘lossy’ rules produce identical *holds* fluents *for this input*.

‘lossless’ optimised Core eFLINT

‘lossy’ optimised Core eFLINT

```
in((add,created(I)),S) :- in(( create,I),S).
in((   holds,I ),S) :- in((created,I),S).
in((   holds,I ),S) :- in((derived,I),S).
in((   enabled,I),S) :- in(( holds,I),S).
```

```
in((add,holds(I)),S) :- in(( create,I),S).
in((   holds,I ),S) :- in((derived,I),S).
```



are hand-crafted and very straightforward: we remove rules which we know will never apply (*e.g.*, because the input produces no violations) and we remove rule-antecedents which we know are always true (*e.g.*, because the input produces no obfuscations).

In the ‘lossless’ case, we restrict this optimisation such that the resulting output is entirely identical. In the ‘lossy’ case, we let the result remove *create* and *enabled* fluents, but fluents with other attributes (*e.g.*, *holds*) remain identical. This result suggests that the output-preserving ‘lossless’ optimisation of the Core eFLINT rules achieved no speedup whatsoever; in fact it was slightly slowed down. But the ‘lossy’ optimisation, which removed uninformative fluents, achieved a significant speedup.

From this, we conclude that Clingo was already performing significant optimisations internally, so our hand-crafted rule- and antecedent-removal optimisations resulted in no speedup. However, there was speedup in the case we reduced the burden on Clingo to produce output. We conclude that (only) more elaborate transformations of the input to Clingo have the potential for speedup. These findings also support our conclusions from Experiment 2.9; Clingo’s optimised choice of derivation rules is a major contributor to the performance speedups our new implementation.

## 2.10 Related & Future Work

**Optimised Usage of Clingo** Our usage of Clingo is conservative, in the sense that the majority of our translation pipeline depends on a minority of Clingo-specific features. In the future, we are interested in augmenting our translation pipeline to optimise Clingo’s performance in reasoning about our scenarios. Some features promise to open the way for new applications of eFLINT specifications, hopefully with minimal changes to our translation pipeline. We consider starting by understanding Clingo’s advanced features. For example, the Clingo input language offers the following ‘meta-statements’ (macros and pragmas): 1. `#show` filters output terms, 2. `#include <incmode>` opts into incremental reasoning mode, such that program parts distinguished by `#program` can be incrementally evaluated and inspected, and 3. `#maximise` lets Clingo solve optimisation problems, which generalise the usual answer-set search. Clingo and the underlying components such as *Clasp* [GKS12] and *Gringo* [GHK<sup>+</sup>15, HLY13] also offer a myriad of configuration parameters which we have yet to explore. For example, in the Clasp solver, 1. `--pre` pre-processes the input ruleset, 2. `--enum-mode` selects from one of several internal solving algorithms, and 3. `--project` defines a projection on output facts, and enumerate only one solution per projection-equivalence class. These features are documented alongside many others in Clingo’s user guide<sup>5</sup>.

---

<sup>5</sup>Available at <https://github.com/potassco/guide/releases>

**Efficient Chronological Reasoning** In our experiments, we found that the run time of both interpreters was superlinear with the length of the input scenario. In principle, we see potential in explicitly exploiting eFLINT’s ‘causality’ property, which we have formalised as all rules being state-internal (Definition 2.2) except Rule (INERTIA), which lets fluents in the  $N^{\text{th}}$  depend only on fluents in the  $N^{\text{th}}$  state and the  $N - 1^{\text{st}}$  transition. Intuitively, any eFLINT interpreter should be able to reason about states and transitions, interleavedly, one step at a time, avoiding the overhead of reasoning about unrelated states. We consider two approaches to implementing this optimisation in our new interpreter. Firstly, we will investigate leveraging Clingo’s *incremental* and *multi-shot* solving features, which are intended for these situations: they let Clingo break up programs, such that rules are evaluated in the order that respects causality. Secondly, we will investigate projecting large eFLINT scenarios into local fragments, *e.g.*, focusing on just one transition at a time.

Unfortunately, these optimisations seem incompatible with our new scenario search usage, where search criteria are encoded via non-state-internal rules; *i.e.*, they may reason back and forward over the scenario. Fortunately, we expect many complex search problems to be expressible via only state-internal rules, *e.g.*, by ‘threading’ bookkeeping information via fluents between states using Rule (INERTIA) as usual.

**Evaluating Scenario Search** We have left the evaluation of our interpreter’s scenario-searching out of scope. We want to more precisely characterise what expressive power its gives users, and then compare this to (model-checkers of) similar languages: Symboleo [PRR<sup>+</sup>24] and Fievel [VC08,SHH24]. How does eFLINT compare in this usage? What can we learn from their approaches to model checking?

**Substituting Clingo** In this work, we prioritised the cohesion in our semantics across the two use-cases of eFLINT. But in the simpler use case of scenario-checking, we make little use of Clingo’s more advanced answer-set solving features. In the future, we are interested in specialising the translation pipeline to target languages more suited to this specialised task. For example, we consider instead targeting the Datalog-based *Soufflé* language [JSS16]. Unlike Clingo, Soufflé has eFLINT-like record types and type checking. Can we translate between these notions of type? Does this result in more efficient reasoning and more explainable output?

**Fundamentally Different Ways to Define eFLINT** The impetus for our work was the observation that, *in abstracto*, eFLINT specifications resemble logic programs. Thus, we chose to give eFLINT a translational semantics via an existing logic-programming language. But we acknowledge that entirely different approaches are available also. For example, [AVDM05] formalises another language which is also normative and used for case analysis and model-checking. But their formalism is based

on first-order linear temporal logic. Consequently, their work is more self-contained; *e.g.*, readers of [AVDM05] are not burdened with the idiosyncrasies of Clingo. But our approach has the benefit of leveraging the existing Clingo interpreter. If the trade-off changes in the future, newer semantics can always be defined and related to this one, *e.g.*, as was done extensively for the Reo language [JA12].

**eFLINT Simplifications and Variants** Our present approach aims to faithfully reproduce the details of eFLINT. However, we identify cases where we capture semantic features in attributes, but they could instead be user-defined, via types and instances. We want to explore the results of removing these features, *e.g.*, to standardise how concepts are modelled, and to simplify our implementation. For example, users can compensate for the removal of *terminate* (*e.g.*, using *obfuscate* instead), and if each *finite*( $[i_1, i_2, \dots, i_n]$ ) clause is approximated by  $n$  by *derive* clauses, *emit* can be removed, because its instances would coincide with those that *hold*.

Moreover, our approach of formalising semantic rules in Clingo eases our experimentation with the semantics of (core) eFLINT. For example, it is easy to change the priority ordering used to resolve conflicts between effects *create*, *terminate*, and *obfuscate* on the same instances, by tweaking the Core eFLINT rules. In this vein, we want to explore the further stratification of the semantic Clingo rules in the hopes of identifying languages that can ease inter-operability between eFLINT and other similar languages. For example, we hope to establish a common core language between eFLINT and *DCPL* (previously called *DPCL* in [SvBPvE22]), which is developed by our colleagues, and is promising, because of its shared notion of state trace, inference rules, and reasoning with various modalities.

**Logical Theories of Dynamic Systems** As in the original article, which defined eFLINT [vBLvDvE20], in Section 2.4, our semantics draws from work on the event-, situation-, and fluent-calculi. Other works (*e.g.*, [ST06]) thoroughly overview and compare these formalisms. Here, it suffices to acknowledge that our notions of state and action were originally inspired by the event calculus [Sha01], but in our simple case of linear traces, many features of the other calculi emerge also. Like *situations*, our scenarios list actions in the order they happen [McC63, McC02].

**Many-Valued Logics** Prior works have generalised the usual Boolean value domain of logical variables [Olk16b, BFC19, OZ19]. For example, [Olk16b] distinguishes *essential* from *accidental* truth, and distinguishes their semantics under combination with  $\wedge$ ,  $\neg$ , and so on, to capture ‘natural language reasoning’. Regardless, the system can be reasoned about systematically. For example, [Olk16a] axiomatises a first-order fragment of the system, and proves its completeness.

Similarly to these works, we distinguish *attributes* of the same instances to encode many truth values, and our semantic rules define how they interact. Also similarly, we prescribe distinct (modal) interpretations of different values. For example, *actViol* truths are undesirable, because they represent actions taken without permission. We also model a three-valued logic in Core eFLINT in three rules that encode the priority relation between attributes *create*, *terminate*, and *obfuscate*.

**Constraint- vs. Logic Programming** An experimental eFLINT model-checking backend was developed in 2022 by Florine de Geus [dG22], inspired by [PRR<sup>+</sup>24]: model checking Symboleo norms via the nuXmv SMT solver. In turn, Florine’s work inspired our own, but we opted to target Clingo instead of nuXmv, because Clingo’s stable-model semantics aligns more closely with the description of eFLINT. This concerns the distinction between logic- and constraint-programming: only the former requires that truths be *supported* by concrete inference steps, *i.e.*, each truth in the model is the root of a proof tree. Consequently, truth are more explainable in practice. For example, in nuXmv but not in Clingo, any  $X$  is true in the trivial specification, where its truth is unconstrained. But in Clingo,  $X$  is false, because its truth is *unsupported*: it is the consequence of no applicable program rule.

## 2.11 Conclusion

We re-defined the eFLINT normative specification language via a translation to Clingo answer-set solving. We defined a new eFLINT semantics and interpreter.

Our semantics is desirable for its rigour. Notably, we have formalised an idea of the original article [vBLvDvE20]: eFLINT clauses express logical inference rules. In the process, we have identified and corrected a flaw in the existing eFLINT interpreter concerning inference about conditions *by default*, which is well-studied in the logic programming community. Our interpreter reasons by default correctly.

In fact, our interpreter consists almost entirely of the Clingo solver; by translating eFLINT to Clingo, we let Clingo perform our normative case analysis. This approach minimises the gap between the semantics and the interpreter, laying the groundwork for experimentation with the eFLINT semantics by changing the (translated) Clingo rules at various levels of abstraction. Fortunately, (our translations between) these abstractions did not incur the performance overhead that we expected for simple scenarios. In fact, it resulted in substantial speedup in most cases: while it had a slowdown of 49% in the worst case, it achieved a 25× speedup in the best case, and scenarios with significant complexity per state were always substantially sped up. We have laid out several future approaches to mitigating the slowdowns. Finally, our approach generalised eFLINT’s typical scenario case-analysis to scenario *search*.

Via Clingo, our tool can recommend scenarios and model-check scenarios against properties in general, and specification-compliance in particular. These use cases were supported in languages related to eFLINT, but not yet in eFLINT itself.

In the future, we want to continue exploring the opportunities for performance optimisation in both scenario-checking and -searching, *e.g.*, by drawing from ruleset preprocessing practices that are well-explored in the answer-set solving community. We are also interested in investigating the intermediate languages we have identified between eFLINT and Clingo. Our *Core eFLINT* is particularly promising, because its embedding in Clingo makes it inter-operable with any Clingo rules, and because it includes much of the eFLINT semantics, but it is simpler and more abstract. We want to investigate these languages as new translation targets for various purposes, and for their suitability – in their own right – to the use cases of eFLINT: normative specification, automated case analysis, and normative model-checking.

**General Takeaways** The existing eFLINT domain-specific language was improved by reframing it as a translation to an existing logic-programming language. This connected the concepts specific to eFLINT’s domain with the highly abstract world of (first-order) logic programming. Moreover, it let us solve new practical problems with existing logic programming methods and tools. We take away that logic programming has a lot to offer: highly applicable formalisms with well-developed tooling.

## Seaso: A Language for the Controlled Multi-Agent Specification of Data Exchange Systems

### Abstract

Data exchange systems are cyber-physical systems focused on the exchange of valuable and sensitive data across physical and organisational boundaries. More than usual, stakeholders are heterogeneously specialised, and motivated to control system behaviour. Necessarily, stakeholders cooperate in the design and control of the system, but currently, are impeded by the complexity resulting from the involvement of their peers.

We define SEASO, a simple logic programming language, whose programs define how they may be extended. Data exchange systems are modelled as SEASO programs, built incrementally by cooperating stakeholders with various specialisations. For example, legal experts define institutional relationships (like *consent* and *ownership*), and system operators model cyber-physical events and states (like *transfer* and *authorised*). Each of these stakeholders works within the constraints imposed by their predecessors, and imposes constraints on their successors. Importantly, checking these constraints requires only ‘shallow’ static analysis; program extensions must not modify types previously marked as *sealed*.

We show the formalisation of a system, starting from behavioural abstractions common in the literature (like *workflows* and *archetypes*) and ending with a particular trace of cyber-physical events. Our goal is to ease the development of robust data-exchange systems by providing SEASO as a vehicle for the communication of unambiguous observations, requirements, and expectations between cooperating programmers.

**Basis of this Chapter** This chapter presents (the case study application of) the interpreter of the novel SEASO language. This artefact is available at [Est25].

### 3.1 Introduction

*Data exchange systems* are the software and infrastructure facilitating the controlled communication and processing of (often large) datasets between organisations. The necessity of careful control over these systems arises from the value of the data that they exchange. Several projects are underway, driven by the collaboration of academic and industrial partners aiming to facilitate controlled data exchange for various use cases. For some examples, the SIMPL EU project provides secure middleware for communication between data exchanges (see <https://simpl-programme.ec.europa.eu>), the Eclipse Data Connector is a framework for data exchange services (see <https://eclipse-edc.github.io>) and Catena-X is an open source project and ecosystem for organisations in the automotive industry to share data (see <https://catena-x.net>). More specialised terms refer to data exchange organisations or infrastructures with particular characteristics, *e.g.*, designed for particular use cases. For example, (*digital*) *data marketplaces* concern the exchange of data as controlled by user participation in data offers, bids, and sales, while *research data exchanges* disseminate data to researchers as controlled by the data owners' consent.

Each data exchange system (use case) places unique and complex requirements on the system. Just eliciting these requirements requires considerable effort, because they arise piecemeal from different stakeholders concerned with different facets of the system. For example, legal experts assert the compliance of system users to data privacy laws, while system administrators control the distributed infrastructure. Just eliciting these requirements is a considerable research effort, because stakeholders are concerned with their own facet of the system, participate at different times, are driven by ever-changing external pressures, and often come into conflict with one another. For example, medical patients are incentivised to minimise the sharing of their sensitive data, while researchers are incentivised to maximise their access to the same data. To keep everyone satisfied, it is important to continuously identify these conflicts and reach agreement on compromises. For example, data producers and consumers agree on a particular interpretation on the requirements for *consent to process data* in the EU *general data protection regulation* (GDPR) [Eur16], and the mechanism used to represent and communicate the consent of data subjects.

The AMdEX-DMI project continues the AMdEX-Fieldlab project in developing safe and reusable data exchange systems. The University of Amsterdam (UvA) contributes to the project by applying the practices of *model-driven engineering* to guide the co-development of data exchange software and stakeholder requirements on the software's behaviour. For example, the current and possible states of the data exchange system are modelled. Second, requirements are modelled as logical formulae

over the state. Third, inconsistencies are identified as dead ends in the state space, where no next step is possible. These kinds of insights feed back to stakeholders and software developers. The former identify conflicts and re-negotiate compromises. The latter harden the data exchange software against bugs and corner cases.

We have had some success in using the eFLINT language for modelling and specifying data exchange systems. It was chosen for its suitable features; eFLINT is designed to inter-connect institutional norms such as contracts and legal regulations with events in software systems [vBLvDvE20]. Moreover, eFLINT is designed for incremental reasoning, where changes to the model are interleaved with queries and inspection [vBKB<sup>+</sup>21]. For example, we can keep the eFLINT reasoner aligned with changes to (an interpretation of) the GDPR at runtime. However, we observe that eFLINT alone is unsuitable for capturing all the stakeholder requirements to the required extent. The problem is that eFLINT is too flexible; there are no protections against (unintentionally) overturning existing properties of the specification under refinement. Contributors cannot express which details of their requirements are subject to change. Example 3.1 demonstrates how this causes problems in practice (and Section 4.4.3 goes into far more detail in the next chapter).

**Example 3.1** (failed cooperative specification in eFLINT). First, ‘Bailiff’ Bob formalises the need for agents to get consent as an eFLINT duty-type; the details of the eFLINT language and the omitted prefix of this specification are not needed for this demonstration. It suffices to see that Bob has defined a complex requirement.

```
Fact current Identified by time
Fact access Identified by time * accessor * data
Duty get-consent Holder accessor Claimant subject Related to access
Derived from (Foreach access, subject-of:
  // the obligation exists whenever a subject's data is accessed
  get-consent(access.accessor, subject-of.subject, access)
  Where access.data == subject-of.data
  // The accessor violates their duty after 10 time units without subject consent.
  Violated when Not(consents(subject,access)) && current.time > time + 10
```

Second, ‘Data-consumer’ Dan refines Bob’s specification to reflect a change to the state of the system. Dan causes a particular data access event to occur, refining the specification in a way Bob considers acceptable. However, Dan also refines the definition of the `get-consent` duty in a way Bob did not anticipate, trivialising it; Dan suppresses the violation of `get-consent` duties. The problem is that Bob cannot express – and Dan cannot recognise – that only Dan’s first change is acceptable to Bob.

```
+access(5, accessor("Dan"), data("X-Ray Dataset")).
Extend Duty get-consent Conditioned by False.
```

To proceed, below, we make explicit our (meta-level) requirements on languages fit for our purposes: incrementally modelling and specifying data exchange systems. We reflect on Example 3.1, concluding that eFLINT falls short in PART.

Name	Requirement on the Language
WIDE	Models formalise concepts from a many sources at many levels of abstraction, from (abstract) ontologies to (concrete) event traces.
NORM	Models formalise an essential notion of prescriptive norms: they define and distinguish what is <i>true</i> from what is <i>desirable</i> .
DYN	Changes in (knowledge of) systems are captured model composition.
EASY	Models are easily composed and checked for well-formedness.
PART	Modellers can control which properties are preserved by composition.
QUERY	Answering finite queries of finite programs takes finite space and time.
CLEAR	The language is as clear and straightforward as possible.

Decades of research has produced a wealth of formal modelling languages. For example, *logic programming* (e.g. Prolog) defines declarative and imperative procedures for inferring the truth of facts given logical rules. *Normative specification* (e.g., eFLINT) defines dynamic relationships between institutional entities as per legal frameworks that reflect real social systems. *Software modelling* (e.g., Alloy) structures objects with attributes as traversable relational databases.

Several existing languages meet several of our requirements, but like eFLINT, many close matches fall short in PART. For example, Datalog is expressive enough to model data exchange systems as rule sets, but partial models (rule subsets) do not specify which properties should remain invariant under composition. Amy formalises access control as relation  $access \subseteq agent \times data$ , but Bob is paralysed by indecision about Amy’s intentions. Is Amy unaware of Dan or did Amy omit Dan from *access* intentionally? Generally, how can intentions be clarified while minimising the burden on each contributor to understand the contributions of their peers? This is a significant problem in practice, where stakeholders specialise on different facets of the system, work at different levels of abstraction, and contribute at different times.

To that end, we presently contribute SEASO, a language for the piecemeal specification of data exchange systems. SEASO aims to be minimal, combining the features of existing languages. In a nutshell, SEASO is a simple logic programming language for modelling data exchange systems by declaring data types and logical inference rules. Additional type-level annotations give facts special meaning. Firstly, *emissive* types witness undesirable properties of the system being modelled by characterising normative violations. Secondly, *sealed* types cannot be modified by subsequent rules, specifying how models may be extended. Concretely, we contribute:

1. an informal overview and formal definition of SEASO (in Section 3.3), and
2. a case study of applying SEASO: a set of composable SEASO programs formalising notions from the (data exchange) literature, including ontologies, workflows, archetypes, plans, and event traces (in Section 3.4).

Section 3.5 discusses noteworthy aspects of our language and approach in application to the data exchange domain (and beyond) by reflecting on the extent to which our requirements are met. Section 3.6 overviews related work offering alternatives to SEASO, or work that complements SEASO. Section 3.7 concludes with a summary.

## 3.2 Background

This section continues Section 3.1 by elaborating on our application domain and approach. The characteristics of the application domain in Section 3.2.1 motivate our language requirements and design decisions. Sections 3.2.2 and 3.2.3 summarise the foundations of our language semantics; we draw attention to notions we aim to formalise, and work pertaining to our requirements on SEASO: `WIDE` `CLEAR`.

### 3.2.1 Data Exchange Systems

Recall from Section 3.1 that *data exchange systems* are distributed systems developed with a particular focus on controlling and facilitating the storage, processing, and exchange of (often large) datasets. Research on the topic is unified by problems concerning this control and facilitation and, depending on the causes underlying these problems, overlaps with research in the following domains.

*Data spaces* are a concept and research area given much recent attention in the industry. The name refers to abstractions that afford system management and maintenance. Often, the complexity of these abstractions reflects the various, complex relationships between the several organisations controlling the systems' distributed resources (`WIDE`); for example, data is exchanged only between organisations that trust one another. [Ott22] is a book that collates much of the work on this emerging field. Therein, the first chapter ('*The Evolution of Data Spaces*') provides a natural entry point to the historical context and recent trends. Work on data spaces exposes data exchange use cases, classifies systems according to their meaningful commonalities, and shows which system abstractions are used and useful in practice. Work on (*Digital*) *data marketplaces* is similar, but proposes generic approaches to developing smaller systems configured and maintained by specialised consortia. These works emphasise federated control, building trust within consortia by formalising their composed (`PART`) requirements in agreements via common abstractions, such

as *archetypes* [ZCG<sup>+</sup>19], to be automatically monitored and enforced (QUERY), for example, via access control [SMV<sup>+</sup>19].

*Multi-agent systems* are comprised of agents distributed over logical (and often physical) networks, autonomously making and acting on decisions. Research explores the relationships between the behaviour of systems at large, and the behaviour of their constituent agents (PART). Owing to the breadth and longevity of this research field, much recent work falls into sub- and related fields, each exploring specialised systems (WIDE). For example, 1. *machine learning* is relevant when agents learn about their system environment, 2. *game theory* lets systems motivate agents to cooperate, despite other motivations to the contrary, and 3. systems are *cyber-physical* when agents interact with the physical world. Work in these areas informs the inherent properties of data exchange systems, their feasible implementations, and the ways they are used if deployed in reality.

### 3.2.2 Logic Programming

Logic programming languages have seen continuous development for decades. Its longevity is owing to its tradition of formalising concepts found to be very generally understandable (CLEAR) and applicable (WIDE). At its heart, logic programming is about defining rules for *inferring* the attribution of truth values to *atomic formulae* (*‘atoms’*), given the valuations of other atoms. Different semantics admit different combinations of inference rules, and use them in different ways. As a simple example, re-arranging rules alters the program output in Prolog but not in Datalog.

A historically challenging subject is programming *with negation*, where the truth of some atoms is contingent on the falsity of others. Naturally, this allows formalising new notions (WIDE) by enabling *non-monotonic* reasoning, where adding rules can change which atoms are true (DYN). Different semantic treatments of negation were proposed to preserve various notions of *consistency*, e.g., no formula is both true and false. For example *negation as failure* [Cla77] proves  $\neg P$  by failing to prove  $P$ . This notion underpins the extension of the well-known SLD resolution [vEK76] to SLDNF [AvE82]. Literature identifies *stratified* programs which guarantee that a particular straightforward operational semantics produces a consistent result. *Local- and Modular-stratification* [Ros90] include more programs, but are more difficult to recognise and exploit [GL88]. Other approaches accept more relaxed notions of consistency. The *stable model semantics* [GL88] gives each program a set of stable models, each the consistent result of assuming different atoms to be false. The *well-founded semantics* is comparable, agreeing on the value of each atom given by *all* stable models if it exists. Otherwise, atoms are given a third, *unknown* value [VGRS91]. Importantly, many atoms have known values in programs with

no stable models. This reduces the impact of inconsistencies on the sensibility of programs, making them easier to compose (EASY). The well-founded semantics underpins our language, so we describe it further in Section 3.3.4.

*Modal logics* introduce modalities for qualifying truth of atoms. For example, *necessity* and *possibility* afford the formalisation and checking of safety properties. For example, ‘is the recipient of any possible data transfer necessarily authorised to read the data?’. *Deontic logic* introduces modalities of *obligation* and *permission*. These are building blocks for norms (NORM), prohibiting behaviours that act without permission, or fail to act on obligation. These notions are pervasive in legal and organisational regulations. For example, Article 19(1) of the GDPR specifies an obligation (to act): ‘the controller shall communicate any rectification or erasure of personal data or restriction of processing [...]’.

Work on logic programming is relevant to data exchange systems, as it informs the applicability of various logics to the modelling of various systems (WIDE).

### 3.2.3 Normative System Specifications

*Normative systems* are necessarily described or specified by norms [Han94]. [AGN-vdT13] overviews this field and its nomenclature, and contextualises the field and its history, whose concepts include the modelling of ancient and persistent societal phenomena such as autonomy, delegation, rights, and power (NORM). For our purposes, *norms* are specifications. For example, a norm prescribes which of the possible system behaviours are preferred. Often, norms are *enforced*, avoiding or correcting *violating* behaviour in favour of *compliant* behaviour.

Much work is motivated by the applications of norm enforcement. Often, the focus is on leveraging the enforced norms, using them as meaningful abstractions over system behaviour in the, past, present, and future. This requires certainty and consensus on the properties of norms. As such, work overlaps significantly with *formal methods*, for example, in uniformly defining systems and their safety properties as sequences of observations [LAT22]. The use of system abstractions as productivity tools characterises the field of *model-driven engineering* [S<sup>+</sup>06]. For example, [Vil10] engineers system requirements by modelling inter-agent dependencies between agents. We draw inspiration from investigations of cooperative model-development, e.g. [KMT12] finds that 1. language complexity causes friction (CLEAR), 2. specialists must be able to naturally model domain context (WIDE), and 3. unexpected changes cause friction (DYN).

The norms enforced in a system are drawn from various sources (WIDE), depending on the system’s intended usage. *Security policies* are norms that prescribe behaviour that preserves security properties, intended to automatically enforce good cyber-

physical behaviour. For example, ‘only encrypted messages cross the network’. ODRL [Ian07] and XACML [ANP<sup>+</sup>03] define policies for agents’ use of resources; these are particular languages with extensive specifications. Other works propose generic mechanisms to controlling system events; for example, role- [San98] and attribute-based access control [San98] and usage control [AK22] control agents’ access to data, differing in the granularity of the norms and the nature of the access. *Regulatory norms* value behaviour of institutional entities and situations. They define properties and abstractions considered valuable to the society modelled by the system, or the society of which the system is a part. As such, they often embrace reacting to undesirable behaviour, rather than avoiding it (c.f. common specifications of software systems). For example, the processing of private data by an unauthorised party is a violation whose occurrence empowers others to impose punishments. *Legal regulations* formalise laws, or norms codified according to legal paradigms. For example, a norm formalises privacy-preserving data processing as per the GDPR. In practice, norms often draw from various sources at once, or concern interactions between physical (‘concrete’) and institutional (‘abstract’) objects. For example, [GMS06] affords checking if business processes comply to business contracts, and [BL08] defines an access control model based on reasoning about the (legal) purpose of data access.

*Normative specification languages* enable the expression of norms, attributing them formal semantics. Their formal nature makes norms unambiguous and precise, affording clear communication (CLEAR) between humans and machines, alike. For example, 1. two humans come to the same understanding of a norm, and 2. the output of an automated tool that checks that a norm preserves given properties is understood by its users as intended by the tool’s designer.

We are particularly influenced by eFLINT, a language used to formalise a wide variety of norms. [vBLvDvE20] introduces the language, and demonstrates the interplay between its features, namely, 1. modelling knowledge of the domain of discourse as relations, 2. defining logical rules for inferring new knowledge from existing knowledge, 3. defining events and their effects on knowledge when triggered, and 4. distinguishing between compliant and violating states and behaviour. [vBKB<sup>+</sup>21] presents extensions to eFLINT, including new statements to modify the definitions of existing relations. The Symboleo language [SPA<sup>+</sup>20, PSA<sup>+</sup>20] and its prototype implementation [PRR<sup>+</sup>24] are interesting owing to their 1. persistent application to model-checking norms against properties, and 2. specialisation to legal contracts, e.g., offering essential notions of contract law as language primitives, including *Asset*, *Contract*, *Party*, and *Role*. We draw inspiration from the space of normative languages suggested by the above two examples, both in the development of our own language, and in the approach to formalising particular normative concepts. For example,

Section 3.4 defines data types sharing similarities with Symboleo’s notions of *Party* and *Role*. Section 3.6.3 compares these languages to SEASO.

### 3.3 Definition of the Seaso Language

This section defines SEASO, a language for the compositional definition of data exchange systems. Section 3.3.1 uses examples to informally overview SEASO. Sections 3.3.2 to 3.3.4 formally define its abstract syntax and semantics. Finally, Section 3.3.5 fixes the concrete syntax used in this paper.

#### 3.3.1 Informal Language Overview

Those familiar with Datalog or Prolog can understand SEASO, in short, as a logic programming language with 1. terminating, bottom-up reasoning, 2. negative conditions, 3. product-typed predicates, 4. using predicate symbols as atom-constructor functions, 5. distinguishing desirable and undesirable predicates, and 6. program well-formedness criteria that are sensitive to the order of program statements.

Firstly, *declaration* statements introduce sets called *domains* (or ‘data types’). *Definitions* are declarations that also specify the structure of domain elements. *Rule* statements define (*inference*) *rules* inferring the presence of domain elements as a function of the present and absent elements of (other) domains. So domain elements are logical variables whose Boolean truth values are computed by the rules; SEASO programs have no other state as in general-purpose languages such as Python or Rust. Rules can also be understood to define logical implications to be preserved. Thus, each program provides an intrinsic definition of the domain of discourse, *i.e.*, ‘what is’. The link to the world outside the model is extrinsic, provided by the user’s interpretation. For example, a person exists iff the corresponding element is in domain `people`, or a property is valid iff the corresponding element is in the domain `properties`.

**Example 3.2** (formalising a transitive trust relation). SEASO program formalising ‘trust relates parties, and is transitive’.

```
decl party. defn trust(party,party). rule trust(X,Z) :- trust(X,Y), trust(Y,Z).
```

Furthermore, *emit* statements annotate domains as *emissive*. This rudimentary, intrinsic, Boolean valuation is the foundation atop which users build extrinsic valuations. We call elements of emissive domains *emitted* or *emissions*. For our purposes, we conflate emitted with *undesirable* such that the language itself captures (negative) system properties. We say the system is *compliant* (or *correct*) if and only if its formalisation in SEASO has no emissions. The name ‘emit’ is chosen to reflect the anticipated use case where an automated tool projects a program to the set of witnesses of its undesirability which are emitted as output for user inspection.

**Example 3.3** (specifying the undesirability of untrusted parties). Extension formalising ‘every party should be trusted’ or ‘untrusted parties are undesirable’.

```
decl party. defn trust(party,party). rule trust(X,Z) :- trust(X,Y), trust(Y,Z).
----- Extension -----
defn trusted(party). rule trusted(P) :- trust(Q,P).
defn untrusted(party). rule untrusted(P) :- P, !trusted(P). emit untrusted.
```

Finally, SEASO is designed with an incremental, compositional approach to programming in mind. Programmers reason about and control not only what is true and emitted by a given program  $p$ , but what is true and emitted by *extensions* of  $p$ , i.e.,  $p$  concatenated with other programs. To this end, SEASO defines *well-formedness* criteria, which include typically desirable properties such as well-typedness, but also extra constraints under programmer control. Concretely, the *seal* statement *seals* a domain, conditioning well-formedness on the absence of subsequent rules for inferring its values. Roughly, a sealed domain ‘has a fixed definition’ (but not fixed elements).

**Example 3.4** (sealing the definition of relation members). Extension formalising ‘the definition of (un)trusted parties is fixed’.

```
decl party. defn trust(party,party). rule trust(X,Z) :- trust(X,Y), trust(Y,Z).
defn trusted(party). rule trusted(P) :- trust(Q,P).
defn untrusted(party). rule untrusted(P) :- P, !trusted(P). emit untrusted.
----- Extension -----
seal trusted. untrusted.
```

Consider the usage of SEASO as a vehicle for formalising and communicating meaning between programmers. As not all extensions to a program are well-formed (e.g. domain `trusted` is sealed), each program constrains the set of its extensions. Effectively, each program prescribes ‘what is (desirable)’, but also ‘what may be (desirable)’. This controlled, incremental program refinement can continue, passing from one programmer to the next, each extension constrained by its predecessors, and constraining its successors. The following, final example refines the definition of `party` such that it becomes an abstraction over `user`. A new property holds forevermore; Dan is not a party. However, the set of users is otherwise unconstrained.

**Example 3.5** (constraining the party relation). Extension formalising ‘parties are users who are not blacklisted, and Dan is blacklisted’.

```
decl party. defn trust(party,party). rule trust(X,Z) :- trust(X,Y), trust(Y,Z).
defn trusted(party). rule trusted(P) :- trust(Q,P).
defn untrusted(party). rule untrusted(P) :- P, !trusted(P). emit untrusted.
seal trusted. untrusted.
----- Extension -----
defn user(str). blacklisted(user). party(user).
rule party(U) :- U, !blacklisted(U). blacklisted(user("Dan")). seal party.
```

In summary, a SEASO program provides intrinsic definitions for 1. a set of domains and the logical relations between their elements, 2. the distinction between desirable and undesirable elements, and 3. the program extensions that are possible, and, consequently, the properties preserved under program extension. SEASO is named with an abbreviation of its statements: seal, emit, and so on.

### 3.3.2 Meta-Language Notation and Abstract Seaso Syntax

Each SEASO program is a sequence of statements, as per the abstract syntax defined by the grammar in Definition 3.1. The syntax definition includes several (syntactic) sorts and their typical elements used as meta-variables, distinguished with subscripts as necessary. For example,  $a$  is an atom, and  $\{a, a_1\}$  is a set of two atoms, with possibly distinct structures<sup>1</sup>. Definition 3.1 defines  $S$ ,  $A$ , and  $D$  as meta-variables for sets of statements, atoms, and domains, respectively. Thus, for example,  $A \subseteq \mathcal{A}$  is true for each  $A$ , because  $A \in \text{powerset}(\mathcal{A})$  is implicit. We use symbols  $\Phi$  and  $\phi$  for arbitrary set and element meta-variables, respectively. Empty sets are written  $\emptyset$ . Syntactic structures are expanded as necessary, and we use (...) to omit irrelevant substructures. For example,  $\text{rule}(a, \dots)$  and  $\text{rule}(\text{cons}(d, a^*), A_p, \emptyset)$  are rules.

**Notation 3.1** (sequences of meta-language terms).

- We use  $\phi^*$  for elements in the sort of sequences whose elements are of sort  $\phi$ . For example,  $s^*$  is a sequence of statements (i.e., a program).
- Length of sequence  $\phi^*$  is written  $|\phi^*|$ .
- We notate sequence elements as usual, e.g.,  $[a_1, a_2, a_3]$  is a sequence of 3 elements.
- Concatenation of sequences  $\phi_1^*$  and  $\phi_2^*$  is written  $\phi_1^* + \phi_2^*$ .
- Where unambiguous, we de-structure sequences in to concatenated sub-sequences and into elements, and we treat  $\phi$ -sequences as  $\phi$ -sets. For example, the formula  $a^* \subseteq ([a] + a^*)$  is tautological for any atom  $a$  and atom-sequence  $a^*$ .

**Definition 3.1** (SEASO abstract syntax). Let  $\mathcal{S}$  give the syntactic category of SEASO statements. Let the statement sequence  $s^* : \text{list}(\mathcal{S})$  be the SEASO programs.

$c, c_1, c', \dots : \mathbb{S}$	(strings)	$z, z_1, z', \dots : \mathbb{Z}$	(integers)
$d, d_1, d', \dots : \mathcal{D} \subseteq \{\mathbb{S}, \mathbb{Z}\}$	(domains)	$v, v_1, v', \dots : \mathcal{V}$	(variables)

<sup>1</sup>As in the literature like [VGRS91], we name atoms to reflect how the semantic logical variables they identify are indivisible ('atomic'). Syntactically, the atoms themselves may consist of sub-atoms.

$$\begin{aligned}
s, s_1, s', \dots : \mathcal{S} &::= \text{seal}(d) \mid \text{emit}(d) \\
&\quad \mid \text{decl}(d) \mid \text{defn}(d, d^*) \mid \text{rule}(a, A_p, A_n) && (\text{statements}) \\
a, a_1, a', \dots : \mathcal{A} &::= \text{var}(v) \mid \text{dvar}(d, v) \mid \text{cons}(d, a^*) \mid \text{int}(z) \mid \text{str}(c) && (\text{atoms}) \\
S, S_1, S', \dots &:\text{powerset}(\mathcal{S}) \quad A, A_1, A', \dots :\text{powerset}(\mathcal{A}) \quad D, D_1, D', \dots :\text{powerset}(\mathcal{D})
\end{aligned}$$

To avoid confusion in our descriptions, in the sequel, we use terms *rule* and *domain* to describe concepts in the SEASO language, while we use terms *meta-rule* and *relation* to describe concepts in the meta-language, which we use to define SEASO.

### 3.3.3 Static Semantics: Program Composition and Well-Formedness

Only elements of  $s^*$  are syntactically-correct SEASO programs. However, only a subset of these are *well-formed*, because they meet no criteria for *ill-formedness*, which are precisely defined in Definition 3.4. This definition depends on the *type* (in  $\mathcal{D}$ ) of each atom, which is defined in Definition 3.2. Generally, we focus on well-formed programs, which have the following, desirable characteristics:

1. Definitions of domains are unique.
2. If  $d$  constructs an  $N$ -argument atom, then  $d$  is defined with  $N$  fields.
3. Domains occurring in the program are declared; definitions count as declarations.
4. Each atom has a unique *type*. For example, if  $\text{cons}(d, [a_1, a_2, a_3, \dots, a_n])$  and  $\text{defn}(d, [d_1, d_2, d_3, \dots, d_n])$ , the construct has type  $d$ , and each  $a_i$  has type  $d_i$  for  $1 \leq i \leq n$ . SEASO programmers can usually rely on this to type each variable  $\text{var}(v)$  in their program. Otherwise, these can be replaced by  $\text{dvar}(d, v)$  which has type  $d$ , but is otherwise the same.
5. Variables that occur in  $\text{rule}(a, A_p, A_n)$  also occur in  $A_p$ , its *positive antecedents*. Literature including [CGT89] calls a program *safe* iff it has finite truths. Later, we see that safety affords a terminating, bottom-up inference algorithm, so it is prevalent in logic programming languages. However, the terminology is confusing. What [LM03] calls *safety*, [CGST15] calls *range-restricted*, and [Cla77] calls *allowed*. Meanwhile, [CGT89] calls a program *safe* iff it has finite truths.
6. The program contains no  $\text{seal}(d)$  before a rule with a  $d$ -type consequent.

**Definition 3.2** (atom type). In the context of SEASO program  $s^*$  and rule subterm  $\phi$ , atom  $a$  has type  $d$  iff  $\text{type}(s^*, \phi, a, d)$ . Intuitively, the first four meta-rules assign types to atoms based on their outermost structure. Then the meta-rule ARG TYPE moves ‘inward’, inferring from the type of construct  $c$  the types of  $c$ ’s parameters. The other meta-rules infer ‘outward’, extracting type-relations from deep inside rule subterms.

$$\begin{array}{c}
\frac{}{\overline{\text{type}(s^*, \phi, \text{int}(z), \mathbb{Z})}} \text{INT TYPE} \qquad \frac{}{\overline{\text{type}(s^*, \phi, \text{str}(c), \mathbb{S})}} \text{STR TYPE} \\
\frac{}{\overline{\text{type}(s^*, \phi, \text{cons}(d, A), d)}} \text{CONS TYPE} \qquad \frac{}{\overline{\text{type}(s^*, \phi, \text{dvar}(d, v), d)}} \text{DVAR TYPE} \\
\frac{\text{type}(s^*, a, a_d, d) \quad a \in A}{\overline{\text{type}(s^*, A, a_d, d)}} \text{INTO ATOM} \qquad \frac{\text{type}(s^*, [a] + a_p^* + a_n^*, a_d, d)}{\overline{\text{type}(s^*, \text{rule}(a, a_p^*, a_n^*), a_d, d)}} \text{INTO RULE} \\
\frac{\text{defn}(d, d_1^* + [d_n] + d_2^*) \in s^* \quad \text{cons}(d, a_1^* + [a_n] + a_2^*) = a}{\overline{\text{type}(s^*, a, a_n, d_n)}} \text{ARG TYPE}
\end{array}$$

**Definition 3.3.** Let  $\text{subterm}(s, a)$  iff atom  $a$  occurs anywhere inside statement  $s$ .

**Definition 3.4** (ill-formedness). These meta-rules define the disjunct *ill-formedness* criteria of SEASO programs, which are *well-formed* unless they are ill-formed.

$$\begin{array}{c}
\frac{\text{defn}(d, d_1^*) \in S \quad \text{defn}(d, d_2^*) \in S \quad d_1^* \neq d_2^*}{\text{ill}(S)} \text{REDEF} \qquad \frac{s \in S \quad \text{subterm}(s, a) \quad \nexists d : \text{type}(S, s, a, d)}{\text{ill}(S)} \text{UNTYPED} \\
\frac{a = \text{var}(v) \vee (\exists d, a = \text{dvar}(d, v)) \quad s \in S \quad \text{rule}(a', A_p, A_n) = s \quad \text{subterm}(s, a) \quad \neg \text{subterm}(A_p, a)}{\text{ill}(S)} \text{UNSAFE} \qquad \frac{\text{subterm}(s, \text{cons}(d, a^*)) \quad s \in S \quad \nexists d^* : (|d^*| = |a^*| \wedge \text{defn}(d, d^*) \in S)}{\text{ill}(S)} \text{NO DEF} \\
\frac{\text{type}(S, s, a, d) \quad \text{decl}(d) \notin S \quad \forall d^* : \text{defn}(d, d^*) \notin S}{\text{ill}(S)} \text{NO DECL} \qquad \frac{s \in S \quad \text{type}(S, s, a, d_1) \quad d_1 \neq d_2 \quad \text{type}(S, s, a, d_2)}{\text{ill}(S)} \text{RETYPED} \\
\frac{\text{seal}(d) \in s_1^* \quad \text{rule}(a, \dots) \in s_2^* \quad \text{type}(s_1^* + s_2^*, s_1^* + s_2^*, a, d)}{\text{ill}(s_1^* + s_2^*)} \text{RULE BREAKS SEAL}
\end{array}$$

The final ill-formedness criterion (RULE BREAKS SEAL) is a notable design choice; it lets program extension generally *not* preserve well-formedness. This breaks the symmetry between the roles of two programmers cooperating to create a well-formed  $s_1^* + s_2^*$  by each contributing one part. It is natural for  $s_1^*$  to be written first, and then used to inform the writing of  $s_2^*$ . However,  $s_1^*$  and  $s_2^*$  can be written independently if constraints on  $s_1^*$  and  $s_2^*$  are agreed upon ahead of time. For example, if both parts include  $\text{defn}(d, d^*)$ , and so, their rules agree on the structure of  $d$ -atoms. In Chapters 4 and 6, such agreements form the foundation of cooperation. For this reason, well-formedness is decidable from only static analysis: checking types and seals. Cooperators can easily and quickly check whether they have cooperated successfully, and if not, what is to blame. We expect that the creation of large, composite programs will necessitate a mix of approaches in practice; sometimes ill-formedness is avoided

through the prior establishment of agreements, and other times, ill-formedness will be repeatedly encountered and corrected. The examples of Section 3.4 demonstrate this approach: programs are incrementally extended. Section 3.5.4 generalises the approach such that programs are composed of independently-written parts.

### 3.3.4 Dynamic Semantics: Inference and Denotation

Ultimately, the dynamic semantics of SEASO is given by *denotation* ( $\models$ ), a relation defined in Definition 3.7. Whenever  $s^* \models \langle A_t, A_u \rangle$ , we call each atom in  $A_t$  and  $A_u$  *true* and *unknown* in  $s^*$ , respectively. We call atoms that are neither true nor unknown *false*. Effectively, each program partitions atoms over these three *values*.

**Definition 3.5** (substitute atom variables). Given substitution function  $f : \mathcal{V} \rightarrow A$ , let  $sub(f, \phi)$  inductively replace each variable  $v$  within  $\phi$  with atom  $f(v)$ . Precisely, we define three mutually-inductive versions of  $sub$  for atoms, atom-lists, and rules. For brevity, here we call them all  $sub$ . Only the rule-variant is used elsewhere.

$$\begin{aligned} sub(f, v) &\triangleq f(v). & sub(f, [a_1, a_2, \dots, a_n]) &\triangleq [sub(f, a_1), sub(f, a_2), \dots, sub(f, a_n)]. \\ sub(f, int(z)) &\triangleq int(z). & sub(f, cons(a^*)) &\triangleq cons(sub(f, a^*)). \\ sub(f, str(c)) &\triangleq str(c). & sub(f, rule(a, a^*)) &\triangleq rule(sub(f, a), sub(f, a^*)). \end{aligned}$$

**Definition 3.6** (saturate  $\implies$ ). Each *inference small step* ( $\longrightarrow$ ) infers a consequent by substituting rule variables with positive atoms s.t. all antecedents are premises. An *inference big step* ( $\implies$ ) gives the positive atoms reachable by transitive small steps ( $\longrightarrow^*$ ) from  $\emptyset$  s.t. no next small step exists.

$$\frac{\begin{array}{l} s \in S \quad s_r = rule(a, A_{pr}, A_{nr}) \\ f : \mathcal{V} \rightarrow A_p \quad a \notin A_p \quad A_{pr} \subseteq A_p \\ sub(f, s) = s_r \quad A_{nr} \subseteq A_n \end{array}}{\langle S, A_n, A_p \rangle \longrightarrow \langle S, A_n, A_p \uplus \{a\} \rangle} \text{ SMALL} \quad \frac{\begin{array}{l} \langle S, A_n, \emptyset \rangle \longrightarrow^* \langle S, A_n, A_{p1} \rangle \\ \nexists A_{p2} : \langle S, A_n, A_{p1} \rangle \longrightarrow \langle S, A_n, A_{p2} \rangle \end{array}}{\langle S, A_n \rangle \implies A_{p1}} \text{ BIG}$$

**Definition 3.7** (denotation  $\models$ ). This relates each SEASO program  $s^*$  to its *denotation*: a pair of atom-sets  $\langle A_t, A_u \rangle$ , where  $A_t$  is interpreted as the *truths* and  $A_u$  is interpreted as the *unknowns*, which are disjoint by definition. Atoms are *unknown* if they are neither true nor false. ( $\models$ ) is defined via  $\rightsquigarrow$ , which computes a sequence of alternating pairs until even elements of the sequence reach a fixed point.

$$\frac{\overline{\langle s^*, 0 \rangle \rightsquigarrow \emptyset} \text{ ALT BASE} \quad \begin{array}{l} \neg ill(s^*) \quad \langle s^*, n-2 \rangle \rightsquigarrow A_{even} \\ n \% 2 = 0 \quad \langle s^*, n-1 \rangle \rightsquigarrow A_{odd} \\ \langle s^*, n \rangle \rightsquigarrow A_{even} \end{array}}{\frac{\langle s^*, \mathcal{A} \setminus A_1 \rangle \implies A_2 \quad \langle s^*, n \rangle \rightsquigarrow A_1}{\langle s^*, n+1 \rangle \rightsquigarrow A_2} \text{ ALT STEP} \quad \frac{\langle s^*, n \rangle \rightsquigarrow A_{even}}{s^* \models \langle A_{even}, A_{odd} \setminus A_{even} \rangle} \text{ DENOTATION}}$$

From a user’s perspective, the truth or falsity of atom  $a = \text{cons}(d, \dots)$  is meaningful;  $a$  is present or absent in a relation identified by  $d$ , and the program prescribes a Boolean answer to a query of the form ‘Is  $a$  true?’. Unknown atoms arise in programs whose rules afford determination of neither truth nor falsity. Finally, let a program’s *emissions* be its true atoms whose types are emissive.

The dynamic semantics of SEASO is based on alternating fixpoint semantics of well-founded general logic programming, as defined in [VG89], which provides constructive approach to the original definition in [VGRS91]. This semantics has been influential for decades. For example, it was independently defined in several works, as seen in [Prz90]. The notion of *bottom-up inference* underlying ( $\models$ ) in particular, and the well-founded semantics in general, is even more widely known and applied. Readers familiar with Datalog, Clingo, Soufflé, or other languages based on the *stable-model semantics* will find Definition 3.6 unsurprising: true atoms are collected incrementally, by substituting variables in rules for true atoms, and then applying the concretised rules: if their antecedents are satisfied, their consequents become true. However, for the sake of completeness, we give a self-contained account of the dynamic of semantics of SEASO, including an explanation of Definition 3.6.

Meta-rule (SMALL) in Definition 3.6 first defines the *small-step inference* relation ( $\longrightarrow$ ). Together, these model the incremental accumulation of knowledge through the process of deductive reasoning; assuming the truth of all antecedents of a rule, one infers the truth of its consequent. In the literature, this step-wise formulation is called (*immediate*) *logical consequence*. This is much the same as the semantics of Horn logic, whose foundation was laid in [Hor51], and expanded and applied for logic programming in countless works, e.g., [vEK76]. Notably, our inference relation is also *monotonic* with respect to the premises, i.e., rules are not conditioned on the absence of true atoms. Our definition differs from the Horn logic in the same way as [VGRS91]: premises are partitioned into positive and negative parts, e.g.,  $A_p$  and  $A_n$  in SMALL-STEP, respectively. Rule consequents are understood as positive, ergo, only the positive premises grow. Note that SMALL-STEP requires no relation between  $A_p$  and  $A_n$ , so a negative atom may be inferred to be (also) positive. For example,  $\text{rule}(a, \emptyset, \{a\})$  formalises ‘ $a$  is true if  $a$  is false’.

Next, meta-rule (BIG) in Definition 3.6 defines the *big-step* inference relation ( $\Longrightarrow$ ), relating a program and negative premises to all positive atoms inferred by a terminating sequence of small steps. The terminal condition – matching ( $\#A_{p2} : \dots$ ) in BIG STEP – reflects our focus on *forward*-reasoning, which is also called *bottom-up* reasoning: each inference step builds up a collection of truths from consequences of applied rules, using these to concretise the rule set in new ways. This forward-direction is reflected in our *safety* criterion (see Definition 3.4), such that the concrete rules

are computable from the truths computed so far. Note in Definition 3.6 that the existence of a big step requires proving the absence of any remaining small step. Intuitively, the big step exhausts all conclusions from a given ‘negative premise’  $A_n$ . But what is the right choice of  $A_n$ ? Where does it come from?

Intuitively, the *stable model semantics* [GL88] gives a non-constructive answer to this question; it characterises premises that give *consistent* conclusions, i.e., each atom is true *xor* false. Its definition requires little more than Definition 3.6. Let  $A$  be a *stable model* of  $s^*$  iff  $\langle s^*, \mathcal{A} \setminus A \rangle \Longrightarrow A$ ; atoms in  $A$  are true, otherwise false. This semantics is desirable, as it captures *non-monotonic* reasoning: adding a rule can create both truth and falsity. For example,  $a_1$  is true in the only model of one-rule program  $\text{rule}(a_1, \emptyset, \{a_2\})$ , but  $a_1$  is false after adding  $\text{rule}(a_2, \emptyset, \emptyset)$ .

Like many others, the well-founded semantics aims to recreate the desirable properties of the stable model semantics, but avoid one of its undesirable properties: a program can have any number of stable models. For example, the program with only rules  $\text{rule}(a_1, \emptyset, \{a_2\})$  and  $\text{rule}(a_2, \emptyset, \{a_1\})$  has stable models  $\{a_1\}$  and  $\{a_2\}$ . The well-founded semantics attributes exactly one model to each program, such that truths and falsities coincide with those of *all* of its stable models, and the rest (here,  $\{a_1, a_2\}$ ) are unknown. The *alternating fixpoint semantics* of [VG89] provides the well-founded semantics with a constructive definition. Its definition is captured by  $(\rightsquigarrow)$  in Definition 3.7. Mechanically,  $(\rightsquigarrow)$  assigns each program an infinite list of atom sets; successive elements represents a round of big-step inference starting from the premise that an atom is false iff it was not true in the previous round (see ALT STEP). Intuitively, this ‘searches’ for the right negative premise until it reaches a fixpoint, which determines the denotation (see DENOTATION).

For the curious reader, Appendix C details the computation of an example denotation by building a proof tree from the meta-rules in Definitions 3.6 and 3.7, giving a proof-theoretic view of the work performed by a SEASO interpreter.

**Example 3.6** (a simple denotation). This following program simplifies an example from [VG89]. Its denotation is understood as ‘ $a_3$  is true, and  $a_1$  and  $a_2$  are unknown’.

$$[\text{rule}(a_1, \{a_3\}, \{a_2\}), \text{rule}(a_2, \emptyset, \{a_1\}), \text{rule}(a_3, \emptyset, \emptyset)] \models \langle \{a_3\}, \{a_1, a_2\} \rangle.$$

Example 3.6 demonstrates how the well-founded semantics underlying the SEASO semantics computes a definitive denotation for a program, despite it having several distinct stable models  $\{a_1, a_3\}$  and  $\{a_2, a_3\}$ . It also demonstrates one reason the well-founded semantics is desirable: it captures what a program’s stable models have in common. Here, it ‘summarises’ these stable models, which disagree on whether  $a_1$  and  $a_2$  are true or false, but agree that  $a_3$  is true. This has a practical implication: SEASO interpreters ‘make no decisions’: the order in which rules are defined or evaluated never matters, *e.g.*, unlike in Prolog programs. In Section 3.5.7

explains how this lets SEASO programmers reason about rules and domains even if they have limited knowledge of the program. Another reason the well-founded semantics is desirable is that it prescribes meaningful valuations to atoms despite rules over-constraining values. Example 3.7 demonstrates.

**Example 3.7** (a program with truths but no stable model). This program has no stable model, but it still has significant meaning under the well-founded-semantics. Intuitively, the truth of  $a_1$  is robust to the unknowable value of  $a_2$ .

$$[\text{rule}(a_1, \emptyset, \emptyset), \text{rule}(a_2, \emptyset, \{a_2\})] \models \langle \{a_1\}, \{a_2\} \rangle.$$

SEASO programmers are not expected to make intentional use of unknown values. When encountered, there always exists some rule whose addition would repair the inconsistency. For example, after adding  $\text{rule}(a_2, \emptyset, \emptyset)$  to the programs in Examples 3.6 and 3.7, all unknown atoms are removed. Section 3.5 addresses the user perspective. Until then, example programs have no unknown atoms.

### 3.3.5 Concrete Syntax

By extending the abstract syntax in Definition 3.1, we briefly introduce a concrete syntax for SEASO, such that it matches what we used in Section 3.3.1:

1. White space delimits tokens, and commas delimit sequence elements.
2. Domain and variables start with lowercase and uppercase letters, respectively.
3. Statements are prefixed by `seal`, `emit`, `decl`, `defn`, `rule`, precisely corresponding to the statement-constructors in Definition 3.1. Statements terminate in a full stop. For example `decl x.` declares domain `x`.
4. Each construct begins with its domain and ends with the sequence of its argument atoms (in parentheses). For example, `friend(P,Q)` is a construct in domain `friend`.
5. As in Prolog (and others), a rule's consequent is followed by `(:-)`, and then, by a sequence of its antecedents in any order. An antecedent is preceded by `(!)` iff it is negative, for example, consider `(rule sad(P) :- friend(P,Q), !friend(Q,R).)`.
6. Strings and integers are atoms in domains `str` and `int`, respectively. These atoms cannot be constructed from other atoms. However, they may occur in the program text as constants, using the usual syntax. For example `41` and `-41` are integers, while `"String?"` and `"-41"` are strings.

For brevity in the SEASO examples, we introduce some syntactic extensions ('sugar'):

1. We may omit `(:-)` from rules without antecedents, and `(.)` from final statements.

2. Each statement written without an explicit tag implicitly has the same tag as its predecessor. For example, (`decl a. b. c`) shows three declaration statements.
3. We admit comma-separated consequents per rule, representing a rule per consequent, each with all antecedents. Effectively, rules admit conjunctive consequents.

**Example 3.8** (a concrete program’s denotation). Let  $s^*$  be program:

```
defn person(str). user(person,admin). admin(user). flag(). set(flag). seal admin.
decl(auditor). defn checked(auditor,flag).
rule A, F :- checked(A,F). person("Amy"). person("Bob"). set(flag()) :- not set(flag()).
```

$$s^* \models \langle \{ \text{person}(\text{"Amy"}), \text{person}(\text{"Bob"}) \}, \{ \text{set}(\text{flag}()) \} \rangle$$

### 3.3.6 Finite and Unique Denotations in Finite Steps

In this section, Theorem 3.1 ultimately shows that the denotation of any finite SEASO program  $s^*$  is unique, and computed in finite steps. Section 3.5.9 reflects on the utility of this result in implementing SEASO interpreters.

**Notation 3.2** (fixed  $s^*$ ). Throughout Section 3.3.6, we fix an arbitrary, finite SEASO program  $s^*$ . Each meta-rule and relation omits the implicit  $s^*$  argument for brevity. For example, we abbreviate each member  $\langle s^*, n \rangle \rightsquigarrow A$  of relation ( $\rightsquigarrow$ ) as  $n \rightsquigarrow A$ .

**Definition 3.8** ( $d$  has field  $d'$ ).  $d \hookrightarrow d' \triangleq \exists \text{defn}(d', d^*) \in s^*, d' \in d^*$ .

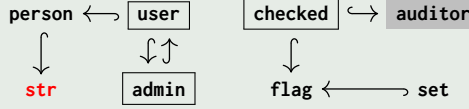
**Definition 3.9.**  $\text{opaque}(d) \triangleq d \notin \{\mathbb{S}, \mathbb{Z}\} \wedge \nexists d^*, \text{defn}(d, d^*) \in s^*$ .

**Definition 3.10.**  $\text{vacant}(d) \triangleq \forall d', (d \hookrightarrow^* d') \implies (d \neq d' \wedge \neg \text{opaque}(d'))$ .

Definitions 3.8 to 3.10 lay the groundwork for a static abstraction over SEASO programs which underlie the proofs to follow. However, these abstractions are also meaningful in their own right. Example 3.9 demonstrates how ( $\hookrightarrow$ ) captures the ontology that is at the heart of a user’s SEASO program. The *vacant* domains are also interesting: their members necessarily cannot be constructed by applying rules because a base case for inductively constructing their elements is missing.

**Example 3.9** (a visualisation of domain vacancy, opacity, and fields). We graphically visualise domains and relations over domains (field, vacant, and opaque) of the SEASO program in Example 3.8. Vacant (opaque) domains are shown in (grey) boxes.

Note how `checked` is vacant, despite having a non-vacant field (`flag`), because another field (`auditor`) is opaque. Note how `user` and `admin` are vacant, despite having no (transitive) opaque fields, because their fields are cyclic. Note how the sub-relation of ( $\hookrightarrow$ ) without vacant domains is acyclic, but not necessarily a connected component.



**Definition 3.11** (the typed Herbrand base). The (*typed Herbrand*) base of SEASO program  $s^*$  is well-defined, inductively on the non-vacant (acyclic) fragment of  $(\leftrightarrow)$ .

$$\frac{s \in s^* \quad \text{subterm}(s, \text{int}(z))}{\text{hb}(\text{int}(z), \mathbb{Z})} \quad \frac{s \in s^* \quad \text{subterm}(s, \text{str}(c))}{\text{hb}(\text{str}(c), \mathbb{S})} \quad \frac{\text{defn}(d, [d_1, d_2, \dots, d_n]) \in s^* \quad \neg \text{vacant}(d) \quad \text{hb}(a_1, d_1) \quad \text{hb}(a_1, d_2) \quad \dots \quad \text{hb}(a_n, d_n)}{\text{hb}(\text{cons}(d, [a_1, a_2, \dots, a_n]), d)}$$

Definition 3.11 introduces an upper bound on the atoms inferrable by rules in a program as a function (only) of its type-definitions; it does not matter which rules are actually included. As suggested by its name, this definition approximates the notion of *Herbrand Base* that is common in logic programming (*e.g.*, it occurs in [Prz90]), but we consider a restriction: it builds only *well-typed* constructs inductively.

**Lemma 3.1** (*hb* is finite). *Proof.* The first two meta-rules in Definition 3.11 have finite applications; they are bounded by the occurrences of integers and strings in the finite program  $s^*$ . The third meta-rule constructs an atom from combinations of *smaller* atoms, as ordered by a sub-relation of  $(\leftrightarrow)$  which is acyclic, because of the  $\neg \text{vacant}(d)$  condition: the chosen domain  $d$  is acyclic in  $(\leftrightarrow^*)$ .  $\square$

**Lemma 3.2** ( $\rightarrow$  stays in *hb*). If  $\langle A_n, \emptyset \rangle \rightarrow^* \langle A_n, A_p \rangle$  then each  $a$  in  $A_p$  is in *hb*.

*Proof.* By induction on the  $\rightarrow$  path so far. The base case of  $A_p = \emptyset$  is trivial. In the inductive step, atoms in  $A_p$  are in *hb*. The step constructs some  $a \notin A_p$  in the selected rule  $s \in s^*$ . Because  $s^*$  is well-formed,  $a$  is well-typed.  $\text{sub}(a, s)$  is the result of mapping variables in  $a$  with  $f : \mathcal{V} \rightarrow A_p$ . Thus  $\text{sub}(a, s)$  is constructed from constants in  $s$  and atoms in *hb*, by the inductive hypothesis. Thus  $\text{sub}(a, s)$  is also in *hb*.  $\square$

**Lemma 3.3** ( $\rightarrow$  is terminating). For each  $\phi \rightarrow^* \phi'$ , there exists  $\phi' \rightarrow^* \phi''$  such that there exists no  $\phi''$  where  $\phi'' \rightarrow \phi'$ ; *i.e.*, each  $\rightarrow$  path eventually terminates.

*Proof.* By Lemma 3.2, each subsequent  $\rightarrow$  step adds a new  $a \in \text{hb}$  to  $A_p$ . By Lemma 3.1, *hb* is finite, so there are at most as many steps as atoms in *hb*.  $\square$

**Lemma 3.4** ( $\rightarrow$  is confluent). For an arbitrary ‘fork’  $\phi_1 \leftarrow \phi \rightarrow \phi_2$ , there exists some ‘join’  $\phi'$  with  $\phi_1 \rightarrow^* \phi' \leftarrow \phi_2$ ; *i.e.*, forking  $\rightarrow$  paths can always join again.

*Proof.* Consider an arbitrary fork. When  $\phi_1 = \phi_2$  the property is trivial. Otherwise, the positive atoms  $(\lambda \langle A_n, A_p \rangle, A_p)$  at  $\phi$ ,  $\phi_1$ , and  $\phi_2$  are some  $A_p$ ,  $A_p \uplus a_1$ , and  $A_p \uplus a_2$ , where  $a_1 \neq a_2$  (and  $\uplus$  denotes the disjoint union). The concretised meta-rule to  $\phi_1$

still applies when  $A_p$  is substituted with  $A_p \uplus \{a_2\}$ , and *vice versa* for  $\phi_2$  and  $A_p \uplus \{a_1\}$ . Hence there exists  $\phi_1 \rightarrow \phi' \leftarrow \phi_2$  with positive atoms  $A_p \uplus \{a_1, a_2\}$  at  $\phi'$ .  $\square$

**Lemma 3.5** ( $\implies$  is a total function). *Proof.* By Definition 3.6,  $\implies$  results from some  $\langle A_n, \emptyset \rangle \rightarrow^*$  where no  $\rightarrow$  next step is possible. By Lemma 3.3, such a path exists for any choice of  $A_n$ . By Lemma 3.4, all such paths have the same end.  $\square$

**Lemma 3.6** ( $\rightsquigarrow$  is a total function). Given arbitrary  $s^*$  and  $n : \mathbb{N} \triangleq \{0, 1, 2, \dots\}$ , there exists a unique  $A$  such that  $\langle n \rangle \rightsquigarrow A$ .

*Proof.* By induction on  $n$ , it suffices to show that there is a unique  $A \subseteq \mathcal{A}$  such that  $\langle n \rangle \rightsquigarrow A$ . In the base case of  $n = 0$ , only ALT BASE in Definition 3.7 applies, so  $A = \emptyset$ . In the inductive case of  $n > 0$ , there is a unique  $\langle n - 1 \rangle \rightsquigarrow A'$  by the inductive hypothesis. Only ALT STEP applies. By inversion of ALT STEP,  $\mathcal{A} \setminus A' \implies A$  witnessing existence of  $A$ . Then because  $A'$  is unique, by Lemma 3.5,  $A$  is unique.  $\square$

**Lemma 3.7** (adding premises strengthens  $\implies$  conclusions). Given  $A_n \implies A_p$  and an arbitrary  $A'_n \supseteq A_n$ , there exists  $A'_p \supseteq A_p$  such that  $A'_n \implies A'_p$ .

*Proof.* Take an arbitrary  $A_n \implies A_p$ . By inversion of BIG, there is some  $\langle A_n, \emptyset \rangle \rightarrow^* \langle A_n, A_p \rangle$  path such that it has no next  $\rightarrow$  step. By induction on the path, it suffices to construct a path with prefix  $\langle A'_n, \emptyset \rangle \rightarrow^* \langle A_n, A_p \rangle$  which has no next  $\rightarrow$  step, and then apply BIG. The prefix is constructed by lifting the given steps by strengthening  $A_n$  to  $A'_n$ , because adding to the negative premises preserves the applicability and result of SMALL. The path is extended by  $\rightarrow$  steps, growing the consequences from  $A_p$  to some  $A'_p$ , but eventually no next step is possible, by Lemma 3.3.  $\square$

**Lemma 3.8** ( $\rightsquigarrow$  conclusions alternatively grow and shrink). For any even  $n : \mathbb{N}$  where for  $m \in \{0, 1, 2, 3\}$ ,  $\langle n + m \rangle \rightsquigarrow A_m$ , necessarily  $A_0 \subseteq A_2$  and  $A_1 \supseteq A_3$ .

*Proof.* By Lemma 3.6,  $A_m$  for each  $m \in \{0, 1, 2, 3\}$  exists uniquely. We proceed by induction on  $n$ . In the base case where  $n = 0$ , only ALT BASE applies, so  $A_0 = \emptyset$ .  $A_0 \subseteq A_2$  is trivial. In application of ALT STEP at  $A_1$ , by the contrapositive of Lemma 3.7,  $A_3 \subseteq A_1$ . In the inductive case, where  $n > 0$  and  $n$  is even, by the induction hypothesis, there exists  $\langle n + i \rangle \rightsquigarrow A_i$  for  $i \in \{-2, -1\}$  such that  $A_{-2} \subseteq A_0$  and  $A_{-1} \supseteq A_1$ . In application of ALT STEP at  $A_0$ , by Lemma 3.7,  $A_2 \supseteq A_0$ . In application of ALT STEP at  $A_1$ , by the contrapositive of Lemma 3.7,  $A_3 \subseteq A_1$ .  $\square$

**Lemma 3.9** (each  $A_n \implies A_p$  implies each  $a$  in  $A_p$  is in  $hb$ ). *Proof.* For each  $\rightarrow$  in the path underlying the  $\implies$  instance, by Lemma 3.2, each  $a$  in  $A_p$  is in  $hb$ .  $\square$

**Lemma 3.10** (each  $n \rightsquigarrow A$  implies each  $a$  in  $A$  is in  $hb$ ). *Proof.* There are just two cases. The case of  $n = 0$  is trivial: only ALT BASE applies, so  $A = \emptyset$ . Otherwise, only ALT STEP applies. There exists  $A_n$  such that  $A_n \Longrightarrow A$ . Lemma 3.9 suffices.  $\square$

**Lemma 3.11** ( $\rightsquigarrow$  reaches an alternating fixpoint). There exists an even number  $n : \mathbb{N}$  and atoms  $A \subseteq \mathcal{A}$  such that  $n \rightsquigarrow A$  and  $(n + 2) \rightsquigarrow A$ .

*Proof.* By Lemma 3.8, successive even choices of  $n$  monotonically grow the atom set. By Lemma 3.10, these atom sets never exceed  $hb$ , which is finite, by Lemma 3.1.  $\square$

**Theorem 3.1** ( $\models$  is a total function). Exactly one  $\phi$  exists where  $s^* \models \phi$ .

*Proof.* By Lemma 3.11, the conditions of the meta-rule DENOTATION in Definition 3.7 are satisfied for some smallest choice of  $n^{\text{th}}$   $\rightsquigarrow$  step. So some  $s^* \models \phi$  exists. By Lemma 3.8, all larger (even) choices of  $n$  prove the same  $s^* \models \phi$ . Then by Lemma 3.6, DENOTATION applies no other way for each choice of  $n$ , so  $\phi$  where  $s^* \models \phi$  is unique.  $\square$

## 3.4 Case Study: Specifying Data Exchange

This section applies SEASO to the domain of data-exchange systems. We formalise models from the data exchange literature as SEASO programs. Fragments 3.1 to 3.16 in the following sections lay out incremental extensions to a single program. Each extension models a facet of the system. Eventually, the abstract system is refined until it captures granular details of a particular system in a particular configuration.

### 3.4.1 Agents Transfer Data

Fundamentally, data exchange systems comprise agents that potentially transfer data.

**Fragment 3.1** (data-exchange system fundamentals).

```
decl agent. data. defn transfer(agent, data, agent).
```

Fragment 3.2 extends using Fragment 3.1 to formalize ‘Alice transfers Sensor Test Data to Bob’. Expressing this fact requires constructing concrete agents and data, which, in turn, requires defining these domains, which were only declared in Fragment 3.1. We expect this to be a common programming pattern; in building a model representing the information in its most concrete form (for example, for testing), undefined types are provided with primitive identifiers.

**Fragment 3.2** (a particular transfer event).

```
defn agent(str). data(str).
rule transfer(agent("Alice"), data("Sensor Test Data"), agent("Bob")).
```

### 3.4.2 Data Computation

A pervasive notion is the modelling of data computation. We see such notions arise in the literature in two senses. Firstly, computations relate data; some data is output and some data is input, or more specifically, some data acts as functions, applied to other data, acting as parameters. Secondly, computations are events that happen, producing outputs from inputs.

Fragment 3.3 shows a very general approach to formalising the ways computations relate data; `computation` is the domain of identifiable groupings of inter-data relations. For example, computations afford a natural interpretation as hyper-edges whose vertices are data. This example serves to demonstrate how logically related definitions can co-exist, despite formalising notions at different levels of abstraction. The most abstract data-dependencies are captured using `involves`, grouping data *somehow* related to computations. Relations `input` and `output` are more specialised kinds of involvement. Finally, `input` is further specialised by `function` and `parameter`. Notably, these abstractions can be mixed, as more general relationships are inferred from more specific ones. Note how the program expresses the logical relations between these domains apparent in its rules. For example, from the program text, one can infer that functions *count as* inputs.

**Fragment 3.3** (data is computed from data).

```
decl computation. defn involves(computation,data). rule C, D :- involves(C,D).
defn input(computation,data). output(computation,data).
    function(computation,data). parameter(computation,int,data).
rule involves(C,D) :- input(C,D). input(C,D) :- function(C,D).
    involves(C,D) :- output(C,D). input(C,D) :- parameter(C,I,D).
```

Fragment 3.4 defines `compute`, formalising computation events performed by agents.

**Fragment 3.4** (agents compute data).

```
decl computation. defn compute(agent,computation).
```

Note that Fragments 3.3 and 3.4 are well-formed when composed in either order, because their extensions are independent. This is still the case if they are each extended with rules. This independence lets two entities cooperate in refining models of multi-faceted systems. Consider a runtime system in which `compute` events produce data as a function of parameters. Two automated services cooperate in building a system model despite working independently; one learns of `compute` events as they happen, and the other learns which parameters are authorised for which computations. Their models are ultimately composed, producing a model that is useful to users because its denotation gives determined answers to sensible questions.

### 3.4.3 Roles of Data in Computation: Workflows and Plans

SEASO focuses on the way programs are extensions of other programs, and are extended themselves. As such, we are most interested in formalising notions ‘in the middle of the abstraction stack’. In Sections 3.4.3 and 3.4.4 we formalise, respectively, two concepts from the literature that fit this description: *workflows* and (*data exchange*) *archetypes*, as in [vdAvH04] and [SVG20], respectively; however, variants of both are described in many other works. In summary, workflows can be understood as models of the system that elaborate on details of computation pertaining to data, but not agents, while archetypes elaborate on agents, but not data.

Workflows are abstract representations of tasks and their inter-dependencies. The term sees use in the literature of various fields, including business processes (e.g., [vdA03]) and cloud computing (e.g., [LZZ<sup>+</sup>19]). With Fragment 3.5, we opt for an essential formalisation of workflows popular in literature [VvEI18]; a workflow is 1. a graph whose vertices are computation *tasks*, and whose directed edges are input-output data dependencies, and 2. yields particular *resulting* data.

**Fragment 3.5** (data computation workflows).

```
decl workflow. defn task(workflow,computation). result(workflow,data).  
rule W, C :- task(W,C). rule W, D :- result(W,D).
```

Fragment 3.5 requires only a small extension to begin modelling *plans*, the attribution of tasks to agents which will perform them. Fragment 3.6 formalises this simple interpretation of planning.

**Fragment 3.6** (workflow planning).

```
defn plan(task,agent). rule T, A :- plan(T,A).
```

### 3.4.4 Roles of Agents in Computation: Data Exchange Archetypes

Fragment 3.7 formalises *data exchange archetypes* as (constraints on) roles agents play in data exchange and computation, while abstracting away the specifics of the data itself. This lays the groundwork for the norms constraining agent actions and relationships shown in Section 3.4.5. This formalises a particular notion of ‘data exchange archetype’, which occurs in various forms in the literature on data exchange systems.<sup>2</sup> Our approach is primarily based on the graphical encoding of Shakeri et al in [SVG20], which is best summarised by a figure, which we include as Figure 3.1. Appendix D shows our formalisation of more archetypes shown in Figure 3.1.

<sup>2</sup>Our effort to formalise these archetypes began in the eFLINT language, and played a large role in motivating the development of SEASO. Nina Verheijen continued the development eFLINT in parallel to this work; the results remain available at <https://gitlab.com/eflint/data-exchange-templates>.

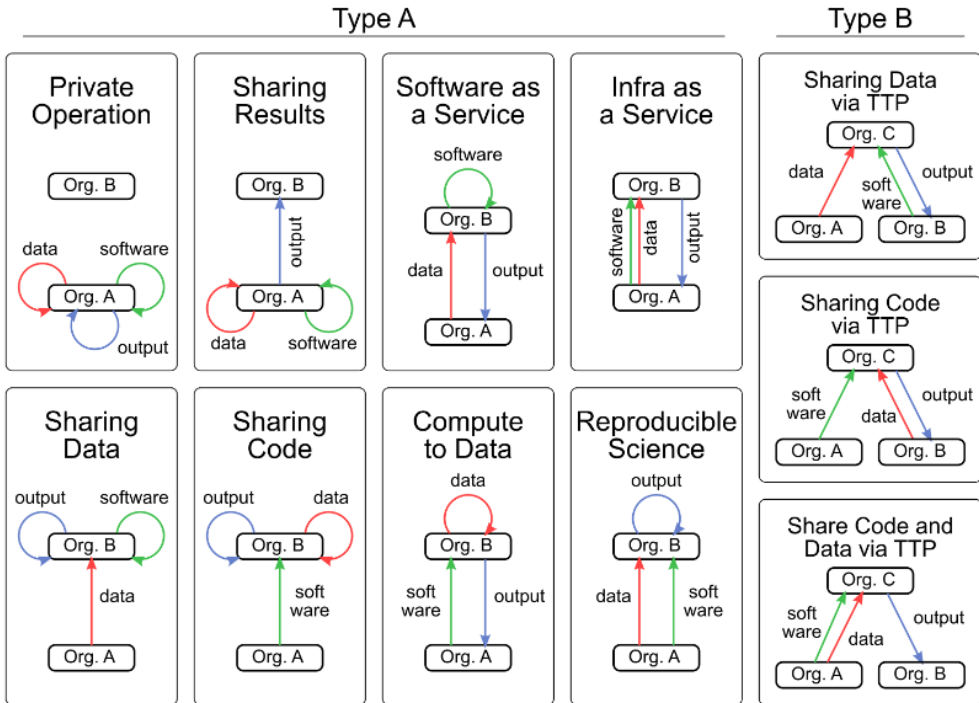


Figure 3.1: Graphical depiction of data exchange archetypes from page 2 of [SVG20]

**Fragment 3.7** (data exchange archetypes).

```

decl context. defn accessor(agent,context). rule A, C :- accessor(A,C).
defn consumer(agent,context). provider(agent,context).
functionProvider(agent,context). parameterProvider(agent,context).
rule accessor(A,C) :- consumer(A,C). provider(A,C) :- functionProvider(A,C).
accessor(A,C) :- provider(A,C). provider(A,C) :- parameterProvider(A,C).

```

Note the similarity in the structure of definitions of inter-data (workflows) and -agent (archetypes) relationships. Firstly, `computation` and `context` are identifiable groupings of entities. Secondly, the relations of each give complementary abstractions of the same events. For example, data playing the role of `function` corresponds with an agent playing the role of `functionProvider`.

Fragment 3.8 gives a slightly different formalisation of archetypes. Concretely, roles are a domain of their own, parameterising the `playRole` relation, which subsumes `accessor` and `consumer`. (Some roles are omitted for brevity.) Inference rules are included to ensure that the model remain conceptually consistent. This demonstrates two useful properties: 1. multiple formalisations of the same concept can be composed, producing a consistent result, and 2. one program may provide multiple, consistent views of one concept. These ease the task of downstream programmers. For example, later extensions may be defined in terms of both `accessor` and `playsRole`.

**Fragment 3.8** (data exchange archetypes v2).

```
defn role(str). playRole(agent,role,context).
rule playRole(A,role("accessor"),C) :- accessor(A,C).
    playRole(A,role("consumer"),C) :- consumer(A,C).
    accessor(A,C) :- playRole(A,role("accessor"),C).
    consumer(A,C) :- playRole(A,role("consumer"),C).
```

Fragment 3.9 adds a new role of `computer` to Fragments 3.7 and 3.8; the `computer` is responsible for performing a (computational) transformation on the data after it is provided and before it is consumed. Precisely, Fragment 3.9 defines a brand new `computer` relation and connects it to the existing `role` relation.

In broader terms, Fragment 3.9 changes the existing archetype abstraction. We expect these kinds of changes to arise in practice as stakeholders learn from each others' fragments, or their requirements change between fragments. For example, this case is explained by the authors of Fragments 3.7 and 3.8 not considering the need for the in-flight computation of data, perhaps because it breaks the symmetry between archetypes and workflows; a `computer` is not associated with any particular data.

**Fragment 3.9** (refining the archetype specification).

```
defn computer(agent,context). rule accessor(A,C) :- computer(A,C).
rule playRole(A,role("computer"),C) :- computer(A,C).
    computer(A,C) :- playRole(A,role("computer"),C).
```

Fragment 3.10 takes a first step in formalising the conceptual connections between workflows and archetypes. Namely, 1. computations are contexts, and 2. the agent performing each compute event plays the role of `computer` in the computation.

**Fragment 3.10** (connecting workflows and archetypes).

```
defn computation(context). rule computer(A,Ctx) :- compute(A,computation(Ctx)).
```

Fragment 3.11 formalises the *compute-to-data* archetype, as it is described in [SVG20]; organisation *A* provides functions to organisation *B*, who applies these functions to (local) data, and returns the resulting outputs to *A*, who consumes them.

**Fragment 3.11** (the compute-to-data archetype).

```
defn orgA(agent,context). orgB(agent,context).
rule consumer(A,C), functionProvider(A,C) :- orgA(A,C).
    computer(A,C), parameterProvider(A,C) :- orgB(A,C).
seal computer. functionProvider. consumer. parameterProvider.
```

Fragment 3.11 shows how program extensions can *raise* the level of abstraction, despite adding detail; here, `orgA` and `orgB` are defined as abstractions over agent roles (`consumer`, `computer` and so on). Subsequent extensions that preserve well-formedness cannot break the seals on agent roles, and so, preserve this abstraction. For example, agents cannot consume outputs without providing functions.

### 3.4.5 Data Exchange Norms

In the examples shown so far, all properties of interest were expressed such that they are necessarily preserved; for example, all computers are accessors. To follow, we explore the formalisation of norms, requiring the expression of properties whose violations are detected and emitted instead. Fragment 3.12 defines a very simple norm; some transfer events are data leaks, which are undesirable.

**Fragment 3.12** (data leaks are undesirable).

```
defn dataLeak(transfer). emit dataLeak.
```

Fragment 3.13 formalises a norm connecting several existing notions: for each compute event that happens, its agent must be planned to perform the computation, and play the role of computer.

**Fragment 3.13** (conditioning desirable compute events).

```
defn happens(compute). planned(happens). unplanned(happens). banned(happens).
rule A, C :- happens(compute(A,C)). rule unplanned(H) :- H, !planned(H).
rule planned(happens(compute(A,C))) :- happens(compute(A,C)), plan(task(W,C),A).
rule banned(happens(compute(A,computation(Ctx))))
  :- happens(compute(A,computation(Ctx))), !computer(A,Ctx).
emit unplanned. banned. seal planned. unplanned. banned.
```

Note that, by orthogonalising the definitions and inference rules of these violation criteria, users can distinguish them, and (externally) attribute them with distinct meanings. For example, users may consider `unplanned` events preferable to `banned` events. This demonstrates the addition of normative meaning to an existing domain; `computer` now influences which compute events are banned.

### 3.4.6 Dynamic (Knowledge of) Concrete Systems

We finish the section by demonstrating the extension of abstract SEASO programs to model system configurations, and modelling how configurations change over time.

Fragment 3.14 shows a 2-part program extension. With only Part 1, `agent("Amy")` has the role of computer, and has performed `computation(context("DataMarketCtx"))`. Consequently, a `unplanned` element is inferred and emitted. However, when the model is subsequently extended with Part 2, the violation is removed. This demonstrates *defeasible* reasoning, i.e., the falsification of prior conclusions when given new information. In practice, it is useful to formalise conclusions subject to change. For example, an automated agent monitors the behaviour of its peers and assigns them punishments for wrongdoings, based on judgements from partial information; punishments are overturned if further investigation raises reasonable doubt of wrongdoing.

**Fragment 3.14** (removing an unplanned event).

```
----- Part 1 -----  
defn context(str). workflow(str). rule orgB(agent("Amy"),context("MarketCtx")).  
rule happens(compute(agent("Amy"),computation(context("MarketCtx")))).  
----- Part 2 -----  
rule plan(task(workflow("AmyWorkflow"),C),agent("Amy")) :- C.
```

As always, the order of program statements does not alter its denotation. However, programs may nevertheless model inherently chronological information. Fragment 3.15 specifies that an agent violates a workflow if they compute data before one of its inputs is computed.<sup>3</sup> The input-output dependency between computations is encoded as `inform`. The reader may recognise the inspiration from event and situation calculi (compared in [KS94]), where chronological reasoning emerges from rules relating facts modelling events (`happens`) and their ordering (`before`).

**Fragment 3.15** (chronologically ordering events).

```
defn before(compute,compute). inform(computation,computation).  
rule before(X,Z) :- before(X,Y), before(Y,Z).  
inform(X,Z) :- inform(X,Y), inform(Y,Z).  
inform(C1,C2) :- output(C1,D), input(C2,D).  
defn violatesWorkflow(agent,workflow). emit violatesWorkflow.  
rule violatesWorkflow(A1,W) :- before(compute(A1,C1),compute(A2,C2)),  
inform(C2,C1), task(W,C1), happens(compute(A1,C1)),  
task(W,C2), happens(compute(A2,C2)).
```

Fragment 3.15 is formulated to decouple 1. whether compute events happen, 2. the order in which they happen, and 3. the inputs and outputs of computations. These domains may be altered independently, for example, by different, physically-distributed, automated agents. Consider an automated task-planner component, responsible for orchestrating (creating) tasks. Fragment 3.16 is a final extension. It helps the planner to avoid planning computations that `inform` themselves.

**Fragment 3.16** (specifying desirable plans).

```
defn informSelf(computation). rule informSelf(C) :- inform(C,C). emit informSelf.
```

In the above examples, we see how data exchange systems are incrementally modelled and specified. At each step, the model can be reasoned about analytically (by understanding the rules) or tested (via temporary extensions). Importantly, the properties established by each programmer are preserved by subsequent extensions, and in the system at large. For example, one may rely on the entire system adhering to the compute-to-data archetype even without reading Sections 3.4.5 and 3.4.6.

<sup>3</sup>This formalisation fundamentally requires that computations are orderable, which is unsuitable for streaming computations. Streams can be modelled via this abstraction as workflows consisting of several discrete computation-steps, but other formalisations may be more convenient for streams.

## 3.5 Discussion

This section reflects on SEASO, discussing its properties and applicability to the formalisation of data exchange systems. Sections 3.5.1 to 3.5.7 use examples to show noteworthy programming patterns, and weaknesses and strengths of the language. Throughout, observations address our requirements of SEASO, defined in Section 3.1. Section 3.5.8 overviews these observations, summarizing the extent to which each requirements is met. Finally, Section 3.5.9 describes our prototype SEASO interpreter.

### 3.5.1 Reasoning about Rules

Each SEASO rule can be read as its corresponding logical implication in *first-order predicate logic* (FOPL); the correspondence is apparent ( $\overline{\text{CLEAR}}$ ). For example, ‘all brothers are friends’ can be encoded as FOPL formula  $\forall A, B : \text{friends}(A, B) \leftarrow \text{brothers}(A, B)$ , and encoded as SEASO statement `rule friends(A,B) :- brothers(A,B)`. Whether reading or writing programs, one can rely on the program denotation preserving the FOPL formula encoded by each rule, regardless of any other statements ( $\overline{\text{PART}}$ ). Example 3.10 demonstrates a system for which this rule-based approach is natural; the program is comprehensible despite its complexity.

**Example 3.10** (modelling integer operators). Modelling operators  $\{+, \times, -\}$  over integers given `suc`, the `int`-successor function.

```
defn suc(int,int). sum(int,int,int). mul(int,int,int). sub(int,int,int).
rule A,B :- suc(A,B). sub(A,B,C) :- sum(C,B,A). sum(A,B,C) :- sum(B,A,C).
    sum(0,A,A) :- A. sum(A,B,C) :- sum(D,B,E), suc(D,A), suc(E,C).
    mul(A,0,0) :- A. mul(A,B,C) :- mul(A,D,E), suc(D,B), sum(E,A,C).
seal sum. sub. mul.
```

Many useful properties can be formalised in FOPL. All properties characterising finite relations can be encoded as SEASO via rules constructing their elements. However, not every FOPL formula can be enforced in an existing program. Those that can are finite FOPL implications whose 1. consequents are conjunctive atoms, 2. antecedents are conjunctive literals, and 3. quantified variables are subterms of positive antecedents. Moreover, they are in *Skolem normal form*, i.e., all variables are universally quantified and at the root of the formula. An arbitrary formula can be *Skolemized* to produce fresh ‘helper’ domains and a set of formulae in Skolem normal form. In practice, Skolemization is straightforward, and its results are easy to understand ( $\overline{\text{CLEAR}}$ ). For example, in Section 3.4.5, defining `unplanned` as ‘the compute events for which there exists no plan’ required Skolemization, producing helper domain `planned`.

SEASO programmers express falsity by not inferring truth. As such, these properties cannot be added to arbitrary existing programs; some specify that truth be preserved by extension ( $\overline{\text{PART}}$ ). For example, given `rule a()`, no extension can enforce  $\neg a$ , but this property can be captured by a fresh atom, for example, by adding `rule na() :- !a()`. The disjunction in  $a \vee b$  cannot be enforced either, but it also cannot be captured by a single rule. However, disjunction can be encoded via several rules: `rule ba() :- a(). ba() :- b()`. Emissive domains model violated properties.

Some conceptually simple notions are cumbersome to express in FOPL, and so also in SEASO ( $\overline{\text{WIDE}}$ ). For example, in Example 3.11, the `count` captures the number of elements in domain `item`. This requires an inductive definition via an order on items, here, provided by `next`. We do not intend for SEASO to be a general-purpose language, so we hesitate to simplify this usage by complicating the language definition ( $\overline{\text{CLEAR}}$ ) with features such as inbuilt counting operators. For now, we leave these features to the discretion of implementors. Section 3.6.1 briefly addresses related work on logic programming, which includes investigations of various language features.

**Example 3.11** (counting items). This program counts items, i.e.,  $\text{count} = \{|\text{item}|\}$ . It assumes `suc` and `next` are successor functions (transitive reductions of total orders) on integers and items, respectively, where  $|\text{next}| \leq |\text{suc}|$ . The last line is an example extension that replaces `count(0)` with `count(2)`.

```
defn suc(int,int). sum(int,int,int). mul(int,int,int). sub(int,int,int).
rule A,B :- suc(A,B). sub(A,B,C) :- sum(C,B,A). sum(A,B,C) :- sum(B,A,C).
    sum(0,A,A) :- A. sum(A,B,C) :- sum(D,B,E), suc(D,A), suc(E,C).
    mul(A,0,0) :- A. mul(A,B,C) :- mul(A,D,E), suc(D,B), sum(E,A,C).
seal sum. sub. mul.
```

### 3.5.2 Reasoning about Atom Values

Generally, determining the truth or falsity of an atom in a program requires computing the (full) denotation. However, in some cases, it is possible to determine atoms' values based on a subset of the program statements ( $\overline{\text{PART}}$ ). In the simplest case, the truth of an atom simply follows from being stated as a fact. Facts are rules and are therefore independent of other program statements (as discussed in Section 3.5.1). A step harder is determining the truth of consequents of rules with only positive antecedents, whose truth must first be determined, inductively.

Determining the *falsity* of an atom requires determining the absence of any rules that result in its truth. If the entire program is not available, falsity is generally indeterminate. This is the consequence of a language feature; atom values are generally subject to change in program extension (as discussed in Section 3.5.3). However, knowing that domains are sealed, one can infer that a wide range of statements

are absent from extensions ( $\boxed{\text{PART}}$ ). For example, consider a program whose first statement is `seal t`; certainly, all atoms in  $t$  are false. In other cases, falsities are determined by analysis of rules and seals. For example, let  $s_1^*$  be the composition of Fragments 3.1 to 3.13; then as `planned` and `banned` are sealed (in Fragment 3.13), each  $s_1^* s_2^*$  preserves  $\forall H : \neg \text{planned}(H) \vee \neg \text{banned}(H)$ . As with visibility qualifiers in other languages, sealing can be used to communicate essential system characteristics, or to establish abstractions for automated tooling to protect.

SEASO programmers are not expected to make intentional use of unknown values. However, we expect them to arise unintentionally when programs are assembled by programmers that have incomplete knowledge of their peers' contributions. Section 3.3.4 defines when atoms have unknown value. Here, it suffices to say that they arise from logical incongruities. Fortunately, unknownness propagates through atoms only to a limited extent, and these do not disturb the truth or falsity of known atoms, nor interfere with the preservation of logical implications between known atoms ( $\boxed{\text{PART}}$ ). Furthermore, for each unknown atom  $a$ , some rules exist whose addition 'repairs' its unknownness; in the simplest case, the domain is unsealed, so (`rule a .`) suffices. Otherwise, unknown values are understood as identifying fundamental, logical incongruities in the specification and the interpretation of its rules as preserved FOPL formulae. Unknown atoms afford a starting point in tracing source of the inconsistency through rules, from consequents to antecedents.

### 3.5.3 Reasoning about Time and Dynamic Systems

Recall that statement order is irrelevant to a program's denotation, affording more significant reasoning under (de)composition ( $\boxed{\text{PART}}$ ). However, we intend large programs to be programmed incrementally, i.e., *design-time* is advanced through program extensions. Thus, program properties can be interpreted as temporal properties of objects over design-time ( $\boxed{\text{DYN}}$ ). For example, property ' $x$  is true *now*, but possibly false *after* further design steps' is expressed via program `defn x()`, while extensions `rule x()` and `seal x` necessitate that `x()` is forevermore true and false, respectively. Note that 'forevermore false' is not expressible without sealing.

SEASO programmers may leverage the expression of properties over design-time to model properties over *system time*, i.e., properties changing in the system being modelled ( $\boxed{\text{DYN}}$ ). We draw attention to two approaches. Firstly, programmers conflate design- and system-time such that program extensions capture changes in (knowledge of) the system. For example, add `rule user("Bob")` when user Bob is registered, or add `rule time(5) :- reach(5), not reach(6). reach(5)` to set `time(5)` such that it can be replaced with `time(6)` by adding another rule later. Secondly, programmers may model several system states in each program, as atoms (much as we formalised states in Core

eFLINT in Section 2.4 in the prior chapter). On one hand, this affords the expression of powerful temporal properties via the usual rules, as demonstrated by Example 3.12. On the other hand, this approach relies on programmers externally attributing the same temporal interpretation to programs, domains, and atoms.

**Example 3.12** (modelling temporal systems and properties). Program chosen to encourage an interpretation of system states (`state`) and temporal precedence (`precedes`), such that `mayEnd` is a temporal property of states with descendants with property `end`.

```

decl state. defn precedes(state, state). end(state). mayEnd(state).
rule mayEnd(S1) :- precedes(S1, S2), end(S2).

```

### 3.5.4 Reasoning about Domains and Signatures

SEASO uses domains as a projection of all a program’s atoms, and admits statements that annotate domains with special properties. For example, `emit x` is comparable to `rule emits("x")`, but only the former is given special treatment by the language definition. Effectively, these are a special class of domain-predicates only stated as facts. Concretely, these properties are defined for a *signature*, a projection of a program which omits all rules, except the domains of their consequents. Reasoning about signatures avoids inference altogether. Figure 3.2 gives a graphical depiction of three program signatures. Observe how signatures have natural interpretations as ontologies (`CLEAR`), defining essential *has-a* (or *composed-of*) and *counts-as* (or *instance-of*) relations over domains. For example,  $s_3^*$  specifies that each *permit* *has a* request and each request *counts as* a permit. By design, signatures suffice to check well-formedness (`EASY`). Again, consider Figure 3.2. The given programs  $s_{1-3}^*$  are each well-formed. Composite programs  $s_1^* s_2^*$  and  $s_2^* s_3^*$  are also well-formed. However,  $s_1^* s_2^* s_3^*$  is ill-formed because `permit` is given conflicting definitions in  $s_1^*$  and  $s_3^*$ . Also note the extensions that preserve well-formedness; 1. `request` is defined repeatedly, 2. `request` is marked emissive after it is sealed, and 3. `access` is marked emissive and sealed before it is defined.

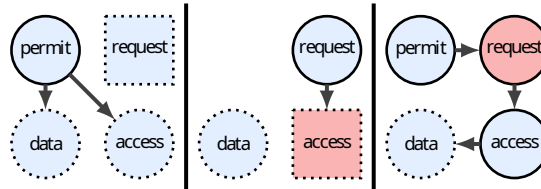


Figure 3.2: Signatures of three SEASO programs  $s_{1-3}^*$ , shown left to right, separated by bars. Domains are shown as vertices. Vertices of declared but undefined domains are dotted, and are related to their parameters by outgoing edges. Vertices of sealed and emissive domains are squares and red (otherwise blue and circles), respectively.

### 3.5.5 Composition Control with Seal Statements

So far, we have focused on a particular approach to building composite SEASO programs; programs are incrementally extended by programmers aware of the program so far, but ignorant of further extensions. Here, we briefly consider the use of two simple language extensions. Chapter 4 focuses on exploring these kinds of features.

Firstly, we consider the introduction of `open` as the dual of `seal`. The ill-formedness meta-rules are extended such that  $ill(s_1^* \text{ open}(d) s_2^*) \leftrightarrow ill(s_2^* \text{ seal}(d) s_1^*)$ . Intuitively, constraints propagate in both directions over the statements, such that each programmer works ‘in the middle of the abstraction stack’.

Secondly, we consider relaxing programs’ total ordering of their statements, e.g., in favour of a *preorder* (transitive and reflexive). This is complemented by the adoption of a more traditional module system to group statements and organise their inter-dependencies. For example, programs are assembled from named *parts*, which each 1. contain a set of statements  $S$ , and 2. name the other parts whose rules do not break (i.e., are excepted from) seals in  $S$ . For example, the programmer of each part  $p$  names only the parts whose rules they are aware of at the time of writing  $p$ . Seal statements are used to protect abstractions in SEASO programs under extension. We see promise in exploring the relation between seals and module systems. We are particularly inspired by a recent work developing a robust and modern module system for Datalog [KPE24], because Datalog and SEASO have much in common.

### 3.5.6 Finite Products with Infinite Extensions

Many examples have shown the utility of defining rules over domains whose parameters remain undefined (e.g., `suc` in Example 3.11). Effectively, undeclared, unsealed domains model arbitrary sets to be defined later. This paradigm is familiar to (meta)-programmers; each SEASO program is a *template* instantiated by an extension that defines its undefined domains. This paradigm is also familiar in norms and regulations ( $\overline{\text{NORM}}$ ), where specifications intentionally incorporate undefined *open-textured terms* [GIM<sup>+</sup>18]. For example, ‘data processing must be justified by *reasonable use*’, leaving room for judgement in a courtroom, on a case-by-case basis. This decouples the programming concerns of 1. defining the structure of atoms in  $d$  from 2. defining relations over atoms in  $d$ . For example, `rule A :- f(A)`, infers  $A$  despite its undefined internal structure.

When extending a program, the programmer is tasked with solving a search problem; what is a satisfactory or optimal extension? The constraints arise from striving to faithfully model the outside world, perhaps a cyber-physical system, or perhaps concepts in their mind. Other constraints are suggested by the program.

For example, which extensions avoid or minimise emissions? SEASO itself assists the programmer insofar as it evaluates candidate extensions, i.e., reveals their broken seals and emissions. In the future, we anticipate more extensively automating this search procedure. Many research areas offer solutions to such search problems, differing in how they represent problems, represent solutions, and traverse the search space. *Answer set solving* is a natural choice; these solvers are given logic programs whose rules describe the traversal of a search space. Unlike SEASO, these rules are not confluent, i.e., atom values are determined by which inference steps are *chosen* over others. For example, given truth of  $c$  and rule  $\{a, b\} :- c$ , one could infer  $a$  is true (but not  $b$ ) or  $b$  is true (but not  $a$ ). Clingo [GKK<sup>+</sup>11] is an example of an established answer set solver; see Section 2.3 for more details. Other fields offer solvers of their own search problems. For example, tools exist for model-checking linear temporal logic properties of nuXmv [CCD<sup>+</sup>14] and Maude [CDE<sup>+</sup>03] programs. With some clever encoding of the problem, these solvers' search for property counterexamples can be used to identify satisfactory or optimal program extensions.

Recall the syntax of SEASO, and its well-formedness criteria from Section 3.3.2. User-defined domains are algebraic *product types* (or *records*), whose atoms are combinations of other atoms. Thus, atoms with the same domain also have the same structure. This has two advantages. Firstly, domains naturally model  $n$ -ary relations; relations are broadly applicable, easily understood, and ubiquitous in the literature and in other specification languages (`WIDE`). Secondly, it ensures that all programs have a finite denotation. Section 3.3.6 argues this result. This is essential to ensuring queries are answerable using finite time and space (`QUERY`). For example, recursively defined domains necessarily have no elements. For instance, consider `defn f(f). rule f(F) :- F` and all its possible extensions.

The disadvantage is that an inherently *polymorphic* set cannot be modelled by a single domain. For example, we cannot extend Fragment 3.4 with a definition of domain `event`, the union of domains `compute` and `transfer`. We are interested in variants of SEASO that enable some kinds of polymorphism. For example, consider a variant of SEASO that removes meta-rule UNTYPED from Definition 3.4, i.e., type definitions of constructs become optional. In this language variant, program rules may arbitrarily construct atoms of undefined types. Finiteness of the denotation is restored by requiring that types of constructs are *less* than types of all their arguments w.r.t. some well-order. Intuitively, this limits the size of atoms.

### 3.5.7 Interpretation of Atom Valuations

*Truth* is the fundamental, intrinsic valuation SEASO programmers use to model systems and theories. Which set of atoms a program deems true is meaningful. By

organising atoms into the potential domains  $\mathcal{D}$ , different programs may organise the same data differently. For example, consider the programs `rule a(1). a(2)` and `rule a(1), b(2)`. The distinction between atoms' domains is intrinsic, giving a starting point for users to attribute accordingly different, extrinsic meanings to atoms' truths. For example, Section 3.4.5 defines `happens` and `planned`, which are most naturally used descriptively ('I observe this'), and prescriptively ('I assert this'), respectively; however, this distinction is external to the program, and so, leaves room for ambiguity and argumentation. Inevitably, there is always some horizon beyond which external interpretation is necessary.

*Emission* is a secondary valuation of atoms, particularly motivated by the need to model normative facets of data exchange systems (`NORM`); emissions are understood as undesirable properties of the system being modelled. Often, the undesirability of a domain is suggested by its name. However, via its definition in the language semantics, emissiveness draws attention to, and prescribes a standard notion of undesirability to the interpretation of atoms in the marked domain. For example, consider the program formalising 'faults are undesirable': `defn bad(fault). rule bad(F) :- F`. Program `emit fault` formalises the same, but via `emit`, its meaning relies less on user interpretation.

We recognise *emission* as an instance, desirable for its simplicity (`CLEAR`), of a more general idea; intrinsically-defined valuations have standard interpretations, and, potentially, special treatment in the language semantics. It remains to be explored which other valuations are worth definition. To follow, we exemplify *flaw* and *stop* as potential valuations of interest, whose intended usages are exemplified in Examples 3.13 and 3.14, respectively. Firstly, `flaw` is identical to `emit`, but its occurrences are understood as the fault of the programmer, rather than faults in the system being modelled; (these are explored in Section 4.4.3.3 where emissions and flaws are called *violations* and *errors*, respectively). Effectively, *flawed* generalises *ill-formed*, with the benefit of being expressed at the atom-level, but with the drawback of being indeterminate given only a program signature. Secondly, `stop` opts into an alternative definition of truth. Intuitively, `stop d` fixes  $d$ 's elements, such that they are unchanged by program extensions. Repeated stops have no effect. Concretely,  $a$  of type  $d$  is true in  $s_1^*$  `stop(d)`  $s_2^*$  iff  $a$  is true in  $s_1^*$ . Thus, (applications of) ordered rules models (occurrences of) ordered events, whose effects outlive their occurrences. A similar result is achieved by instead externalising the encoding of  $d$ 's current elements to the definition of a newly defined and then sealed domain  $d'$ . Consider how the rules defining  $d'$  may be generated from the current denotation.

**Example 3.13** (experimental feature: flaws). This extends Example 3.11, formalizing assumptions of *next* (previously stated in the caption) as *flaws*. For example, extending this program with `rule less(3,3)` results in the addition of flaw `cyclic(3)`.

```

defn eq(item,item). less(item,item). unordered(item,item). cyclic(int).
rule eq(I,I) :- I. cyclic(I) :- less(I,I).
    less(I,J) :- next(I,J). less(I,K) :- less(I,J), less(J,K).
    unordered(I,J) :- I, J, !less(I,J), J, !less(J,I), !eq(I,J).
flaw cyclic. unordered. seal cyclic. unordered. eq. less.

```

**Example 3.14** (experimental feature: stops). Marking a domain `stop` fixes its elements, regardless of future program extensions. It cannot be stopped again. This program, used to extend another that defines some users, fixes the founding users. No subsequent extension can change the founders, even those that add and remove users.

```

decl user. defn founder(user). rule founder(U) :- U. stop founder.

```

### 3.5.8 Requirements Revisited

Here, we collate our reflections on the fulfilment of our requirements of SEASO:

Name	Summary of the extent to which SEASO meets the requirement
WIDE	The language is widely applicable, as it is based on ubiquitous concepts: <i>n</i> -ary relations and first-order predicate formulae.
NORM	Emissiveness separates the modelling of systems (as truth) from the identification of normative violations (as emissions).
DYN	Truth of atoms can be added or removed by program extensions, reflecting changes in (knowledge of) systems. Thus, extensible programs model dynamic systems. Each program can also model several states at once, but these interpretations are external.
EASY	Program composition is simply sequence concatenation. Checking the well-formedness of any composite program requires only its signature, i.e., it does not require computing the denotation, performing inference steps, or even instantiating its rules.
PART	Logical implications are preserved by extension. Moreover, the preservation of atoms' truth or falsity under extension 1. can be enforced by sealing, and 2. is unaffected by 'unrelated' logical inconsistencies.
QUERY	Any finite (query in any) program is evaluated in finite steps.
CLEAR	The language is simple, requiring no knowledge of the data exchange domain. The most complex language facets (typing and inference) are based on well-established work.

### 3.5.9 Prototype Implementation

The simplicity of SEASO eases the task of implementing an interpreter, and affords typical optimisations. For example, during inference, 1. as variables are typed, positive atoms can be more efficiently retrieved if grouped by domain, and 2. as all like-typed atoms have the same, finite structure, they have a fixed size, affording efficient contiguous storage and indexing.

To witness the practical realisability of the language, we provide a prototype implementation of a SEASO interpreter; its source code is hosted in a persistent repository [Est25]. Given a program, the interpreter checks well-formedness and prints the denotation. Its implementation reflects the language definition in this paper as closely as possible, for example, in its concrete syntax. The examples provided in the repository include all those throughout the chapter (excluding Examples 3.13 and 3.14 which would require language extensions).

## 3.6 Related work

Sections 3.6.1 and 3.6.2 present literature offering alternatives to SEASO’s technical foundations, respectively presenting alternative logic programming languages and approaches of controlling program composition. Sections 3.6.3 to 3.6.5 present alternatives to SEASO for modelling data exchange systems.

### 3.6.1 Logic Programming Languages

Considering only rules, SEASO is similar to many logic programming languages. We draw particularly from work on Datalog, maximising their similarity. For example, [Lif10] and Section 3.3.4 define their respective stable model semantics. We expect this similarity to ease the application of work on Datalog to SEASO. For example, [DMRT14] constructs *provenance information* for Datalog programs, making them easier to understand by ‘explaining’ the sources of truths.

Some features of logic programs in the literature are intentionally omitted from SEASO. For example, *disjunctive heads* and *choice operator* (explained concisely in [LM03]) afford convenient expression of new properties, but complicate compositional reasoning; atoms are given multiple truth-valuations, and so, answering ‘is *a* true?’ requires context we prefer remain unnecessary to downstream programmers ([CLEAR]). [NO18] presents a variant of the well-founded semantics supporting rules with disjunctive heads, potentially offering means of generalising SEASO’s underlying semantics while preserving its practical applicability.

Some Datalog dialects include features that appear to complement those of SEASO. For example, [LM03] defines a Datalog extension to include various classes of *constraints*, conditioning the application of rules. Some languages are significantly more expressive. For example, [AK16] introduces functional programming aspects to Datalog. The *Curry* language, surveyed in [Han13], further aims to unify functional and logic programming. It remains to be seen which language extensions to SEASO would be worthwhile in practice.

### 3.6.2 Approaches to Consistency-Preserving Program Composition

Several works provide algorithms for extending declarative programs while preserving properties of interest. Like SEASO, these algorithms rely on a given ordering on fragments (e.g. rules), and emphasise the preservation of properties under composition. They differ from SEASO in that their property of interest is always some kind of consistency in the denotation, such as  $\neg(a \wedge \neg a)$ . Many such properties cannot be expressed for a given program in terms of seals. We see these works as offering SEASO programmers new ways to find programs that are desirable for their consistency.

[SBvE15] defines algorithms for 1. translating between rule base representations oriented around value-constraints and rule-priorities, and 2. rewriting rule bases to prefer the consequents of rules with higher priority. This work extensively explores the cases in which threats to consistency arise in practice, and how they can be mitigated via constraints and priorities.

CP-nets (‘conditional *ceteris paribus* preference’ networks) denote a partial *preference* order on a set of *features*. [HPS21] defines the *enrichment* of CP-net  $x$  and  $y$ , producing a composite CP-net  $z$  that preserves existing preference relationships. Like our own, this work is motivated by users reasoning about  $z$  in terms of  $x$  despite ignorance of  $y$ . Their work offers a ‘lossy’ approach; elements of  $y$  are discarded as necessary. In SEASO, this would mean discarding statements from  $y$  that would otherwise break seals in  $x$ .

### 3.6.3 Executable Norm Specification Languages

Various norm(ative) specification languages are presented in the literature, each based on some established legal framework. Their models formalise regulatory norms. These are used, for example, to define and model-check the compliance of a cyber-physical system to contract law.

eFLINT is a normative specification language that was a major inspiration for our work. Both SEASO and eFLINT model knowledge as  $n$ -ary relations, denote the truth of atoms, and distinguish compliance and violation [vBLvDvE20]. The

most noteworthy difference is semantic, pertaining to the meaning of the order of statements. Recall how SEASO only uses statement order to match constraining and constrained statements. In contrast, eFLINT statements have *effects*, denoting a sequence of transitions through a transition system whose vertices each denote a rule- and truth-set. Both languages enable cooperation via incremental program extension, i.e., Amy and Bob cooperate by writing two successive program parts. Intuitively, eFLINT empowers Bob, who enjoys the freedom to arbitrarily update rules and truths, reflecting chronological system state updates. In contrast, SEASO empowers Amy, who enjoys non-trivial control over Bob, and so, can reason about the composite program. Syntactically, eFLINT prioritises readability by legal experts via familiar terms used in familiar ways (such as `Duty`, `Claimant`, `Act`, `Actor`, and `Conditioned by`), whereas SEASO prioritises terseness and similarity to Prolog and Datalog.

Symboleo [PSA<sup>+</sup>22] and FIEVeL [VC08, VC07, SHH24] are other normative specification languages. In lieu of a thorough comparison, here it suffices to say that these are comparable with eFLINT, but differ in specifics of their underlying legal frameworks, their developmental histories, tooling, and intended applications. Each is used to model dynamic, normative systems, providing them formal semantics and laying the groundwork for testing and model-checking properties. At the heart of each language is a core ontology, whose concepts are instantiated and extended by programmers to build particular models. These ontologies overlap significantly, but also differ in complex ways. For example, each includes *event*, only FIEVeL distinguishes between institutional and base *agents*, and only Symboleo groups obligations into *contracts*. Our impression is that FIEVeL is the most focused on model-checking, and eFLINT has the most generic ontology, and the strongest emphasis on representing implicit knowledge as logical inference rules (via eFLINT clauses `Holds when` and `Derived from`, comparable to SEASO rules).

Various works on normative systems influence SEASO. Firstly, works identify concepts worth modelling. For example, [BvdT08] characterises the Deontic nature (i.e., what *ought to be*) of duties, and that duties are obligations, which are regulative norms, which are substantive norms. eFLINT uses `Duty` to model duties, and we ensure SEASO can model duties also (using `emit`). Secondly, works leverage norm specifications for benefit. For example, [vBKB<sup>+</sup>21] inter-relates formalisms of access-control rules and legal regulations by composing eFLINT programs, [LSvE20] enforces compliance of smart contracts to norms modelled by eFLINT programs, [PRR<sup>+</sup>24] model-checks temporal properties in Symboleo programs, and [VC07] does likewise with FIEVeL programs. We aim to facilitate similar usages of SEASO programs in future.

### 3.6.4 Software Modelling Languages

Several languages, presented in the literature, model software systems primarily in terms of logical formulae and relations. We see benefit in further exploring their literature and adapting their tooling. SEASO sets itself apart by focusing on a controlled, cooperative modelling process by internalising constraints over program extensions (via seals).

Alloy has very much in common with SEASO: 1. all data is modelled entirely as members of  $n$ -ary relations, 2. properties of interest are expressed in the model itself, and 3. dynamic behaviour is captured without built-in notions of time or state [Jac19]. We also see similarity in how programs are built and understood. For example, what we call an *ontology* in Section 3.5.4 is called a *classification tree* in [Jac19]. We hope to exploit this similarity in learning from work on Alloy, for example, in translating SEASO programs to (optimised) SAT problems for analysis.

Several modelling languages complement *Unified Modelling Language* (UML), appealing to users of UML, for example, system administrators and software engineers. *C4* [Bro18] defines models with four levels of abstraction (hence the name). Each level expands the internals of components of previous layers, ultimately ending with UML diagrams. *OCL* augments UML class diagrams, constraining objects of classes via expressions over attributes; *invariants* constrain system states, and *behavioural interfaces* constrain the state transitions triggered by method invocations [RG02]. These approaches have in common with SEASO an emphasis on compositional abstractions [VHG20]. SEASO differs by deferring the choice of system representation to the programmer as much as possible. However, we expect that these languages may inform the domains and rules SEASO programmers may use in practice.

### 3.6.5 Goal-Oriented Requirements Engineering

[WdPAOdPL09] compares *i\** and *KAOS*: influential *goal-oriented requirements engineering* and *requirements specification* languages. These languages are used to express (non-)functional requirements indirectly, by representing actors' (soft) goals directly. There is conceptual overlap with Sections 3.6.3 and 3.6.4, including the normative inspiration of the former, and the fixed model layers of the latter.

The *i\** framework was developed in response to software systems increasingly exhibiting properties typical of social systems: more dynamism, less predictable structure and behaviour, etc. [Yu01], and sees continued application [Yu09]. *i\** affords an agent-oriented worldview by modelling relations between agents, goals, resources, and tasks. For example, *composition* relates task-pairs, and *goal-dependency* relates task-goal-task triples. Models consist of layers modelling intra- and inter-agent

relationships. KAOS is similar to the  $i^*$  framework in that it 1. models requirements in terms of actor goals, 2. represented in terms of some prescribed relations, 3. spread over a fixed set of model layers [WdPAOdPL09]. KAOS differs from  $i^*$  in the prescribed relations and the details of how models are layered. Consequently, they afford different abstractions and approaches to model analysis.

Like SEASO, these languages model systems in terms of relations. Furthermore, they emphasise their extensibility;  $i^*$  and KAOS are presented as frameworks customizable for the use case (in [Yu09] and [UBL<sup>+</sup>11], respectively). Effectively, SEASO takes this to the extreme, prescribing no domains in particular, instead prescribing how domains are populated and defined by programmers. Nevertheless, the early introduction of *agent* in Section 3.4 exemplifies the overlaps in our approaches to modelling our overlapping application domains. We also recognise overlaps with our own motivations and intended usages. For example,  $i^*$  and SEASO emphasise leaving system details temporarily unspecified, for example, to model how autonomous agents have unknown behaviour [Yu09], or to leave room for case-specific judgement (see Section 3.5.6). As such, we expect some SEASO programs to resemble  $i^*$  and KAOS in practice. We see value in a rigorous attempt to embed (the precisely-defined relations of) KAOS and  $i^*$  in SEASO.

### 3.7 Conclusion

We define SEASO, and demonstrate its intended usage in modelling data exchange systems. Our approach is typical, in that it empowers users to reason about systems in terms of formal models built from well-understood and -established foundations: logic programming and relational databases.

Our approach is atypical in its emphasis on the incremental development of program models, such that, each step of the way, the program itself specifies how it may be extended. Intuitively, each model is a contract between upstream and downstream programmers. This is motivated by the application domain; data exchange systems are complex and must adhere to the requirements of various stakeholders with heterogeneous specialisations, working at different levels of abstraction. For example, legal experts reason about data privacy laws, while system administrators reason about firewall rules. Each programmer reasons about the set of properties of their program preserved by (arbitrary) extension, and grow this set by *sealing* domains. This lays the groundwork for controlled cooperation; programmers express their properties of interest in terms of seals, and rely on automated tooling to check that all seals are preserved by a given program extension.

SEASO has simple technical foundations, based in the fundamentals of logic programming. This aides the applicability of SEASO to a variety of problems. We have found it to be sufficiently general to express a variety of concepts from the data exchange literature. For example, we have formalised workflows, plans, archetypes, and event traces. More importantly, by formalising these concepts in the same language, they may be meaningfully composed. We explain how SEASO satisfies the requirements we have identified in the application domain, e.g., concerning expressivity and compositional programming and reasoning. Finally, we provide a prototype implementation of a tool that checks the well-formedness of SEASO programs, and computes their denotations.

Our goal is to afford meaningful cooperation in the development of – and reasoning about – complex models of inherently complex data exchange systems. SEASO is a vehicle for the communication of unambiguous meaning between human and automated agents, alike. This includes stakeholders in data exchange systems, despite them working at different levels of abstraction, contributing at different moments in time, and having limited understanding of each others' specialisations.

**General Takeaways** By defining and applying the SEASO language, we have shown that, by adding some simple new language features, essentially familiar specifications become better-suited to a new context where agents cooperatively develop a shared specification in a controlled manner, *e.g.*, to control a data exchange system.



# Cooperative Specification via Composition Control

## Abstract

High-level, declarative specification languages are typically highly modular: specifications are comprised of fragments that are themselves meaningful. As such, complex specifications are built from incrementally composed fragments. In a cooperative specification, different fragments are contributed by different agents, usually capturing requirements on different facets of the system. For example, legal regulators and system administrators cooperate to specify the behaviour of a data exchange system. In practice, cooperative specification is difficult, as different contributors' requirements are difficult to elicit, express, and compose.

In this work, we characterise cooperative specification and adopt an approach that leverages language features for controlling specification composition. In our approach, specifications model the domain as usual, but also specify how specifications may change. For example, a legal regulator defines 'consent to process data' and specifies which agents may consent, and which relaxations of the requirement are permitted. We propose and demonstrate generic language extensions that improve composition control in three case study languages: Datalog, Alloy, and eFLINT. We reflect on how these extensions improve composition control, and afford new data exchange scenarios. We use this insight to develop SEASO and Slick: bespoke languages for cooperative specification. These developments contribute to the greater vision of multi-agent data exchange to the satisfaction of their shared, complex, dynamic requirements.

**Basis of this Chapter** This chapter adapts the SLE2024 conference article [EvB24]. The same definitions are presented more explicitly and with additional discussion and examples. Section 4.5 replaces a prior 'Future Work' subsection with a (re)definition of SEASO and Slick, which are used in Chapter 3 and Chapter 6, respectively.

## 4.1 Introduction

Data exchange systems are large distributed systems, concerned with the controlled exchange and processing of data across organisational boundaries [OSTL19, OtHW22, vBORS<sup>+</sup>24]. Each system is driven by the participation of autonomous *agents*. These include organisations guiding the development and structure of the system ahead of time, or the cyber-physical entities driving system behaviour at runtime. When these systems behave counter to agents' requirements or expectations, agents may be harmed. For example, if the medical records of patient Bob are unintentionally broadcasted to international data processors, (the privacy of) Bob is harmed. It is imperative that these systems be carefully controlled to minimise harm. At the very least, this is a legal requirement in the European Union as per the EU General Data Protection Regulation (GDPR) [Eur16], which formalises the prevention of harm as the compliance to norms, *i.e.*, the prohibition of normative violations.

In this work, we take a typical approach to controlling complex systems in general, and data exchange systems in particular; (the desirable behaviour of) the system is *specified* in some formal language, producing an artefact called a *specification*. This approach clarifies a way for agents to agree on their requirements on the system. First, the agents agree on the specification language. Then it suffices for them to agree which specification captures each of their requirements. A sufficiently precise and expressive language affords robust and predictable reasoning about specifications. Finally, the agents reason about the real system itself via its specification, relying on the *enforcement* of the specification, which maintains the compliance of the system to its specification, for example, by automated enforcer agents.

The focus of this work is not enforcement, whose theory and practice is explored in other works. For example, in some cases, non-compliance is prevented ('ex-ante enforcement'), e.g., employing access control [HKF15, SCFY96], model checking [CHVB18], or by generating implementations from specifications. In other cases, enforcement is instead continuously or periodically monitored, detected, and corrected ('ex-post enforcement'), e.g., employing usage control [SP03, PHB06, JD22], runtime verification [BF18], or process mining [vdAC22]. The focus of this work is also not the ways specifications model physical systems or conceptual domains of discourse. For example, many specification languages differ in their fundamentals, for example, providing different core ontologies from which their specifications are built. For example, both eFLINT [vBLvDvE20] and Symboleo [SPA<sup>+</sup>20] propose an essentially relational worldview, but only Symboleo builds specifications using *Contract* as a primitive concept, while eFLINT instead provides *Fact* and *Event* as primitives, which can be instantiated and composed to model contracts. Moreover,

multiple semantics may be associated with the same syntax, each with different reasoning capabilities and computational complexities, as is the case for the various profiles of OWL2 [W3C12] (Web Ontology Language version 2).

This work focuses on the features of specification languages supporting *cooperative specification*, a process by which cooperating agents systematically develop a shared specification that captures all their requirements. Precisely, the specification is assembled incrementally as agents contribute new parts. The challenges typical to developing specifications are exacerbated by the characteristics of data exchange systems in particular: these systems are subject to complex and changing requirements, as a consequence of the large number and variety in the agents, reflecting the number of participating organisations, and the complexities of their roles in the exchange of data. For example, legal experts formalise institutional entities and their relationships, process orchestrators coordinate the processing and communication of automated workers, while data providers negotiate, refine, and update the conditions on the use of data. For these agents to develop a shared specification, at some point, in some form, the agents must reason about, communicate about, and control each others' contributions to the specification. How are conflicts between requirements identified and resolved? How do agents determine which properties of the specification may be changed? Example 4.1 shows how agents experience a usability problem as a result of failing to regulate each others' contributions to the shared specification.

**Example 4.1** (motivating example). Agents Amy, Bob, and Dan cooperate to develop a shared eFLINT specification which formalises all of their requirements.

'Administrator' Amy begins, contributing the notion of *processing*, formalised in the specification as an agent-data relation. At this stage, the relation has no elements, but Amy intends for this to be changed by subsequent contributions:

```
Fact processing Identified by agent * data.
```

Next, 'Barrister' Bob contributes to the specification, formalising a simplistic notion of *consent* (to the processing of data) as an agent-processing relation:

```
Fact consent Identified by agent * processing.
```

'Data-consumer' Dan contributes an assertion of the membership of particular processing and consent elements in the relations introduced by Amy and Bob.

```
Extend Fact processing Derived from processing("Dan", "Amy's XRays").
```

```
Extend Fact consent Derived from consent("Amy", processing("Dan", "Amy's XRays")).
```

The two fact-type extensions in Dan's contribution are similar from the perspective of the eFLINT semantics: they assert the existence of relation elements. However, the latter extension violates Bob's meta-level requirement that consent somehow only originates at the agent in question; Dan must not grant Amy's consent! The specification fails to sufficiently capture these kinds of meta-level requirements.

Our approach is to leverage specification languages for (homogeneous) *meta*-specification: like Example 4.1, specifications specify domain-level concepts as usual. But unlike Example 4.1, specifications additionally capture the agents' meta-level requirements: how the specification can be changed. We are inspired, in part, by cooperative programming, where it is commonplace for programs to constrain their own extensions. Consider two examples: 1. Java class members marked *private* capture meta-level requirements and harden the program against bugs at compile time, and 2. invariants of Java functions are encoded as assertion statements, which halt program execution whenever they fail at runtime. Prior contributors experience this as control over the specification. Future contributors experience this as insight into which contributions are considered permissible. The homogeneity of our meta-specification approach enables the robust formalisation of requirements inter-relating domain concepts (like *data*) to meta-level concepts (like *agents* and *specification-contributor*). Subsequently, we evaluate existing languages through this lens, and propose language extensions that improve their suitability to cooperative specification.

Precisely, in this chapter, we contribute the following:

1. a definition of *composition control* in a specification language, affording reasoning about its suitability to cooperative specification (Section 4.2),
2. the definition and evaluation of various composition control mechanisms in the Datalog language, which is chosen for its simplicity (Section 4.3),
3. the evaluation of the suitability of realistic specification languages Alloy and eFLINT to cooperative specification, before and after language extensions adapted from those shown in Section 4.3 (Section 4.4), and
4. based on our experiences with Datalog, Alloy, and eFLINT in Sections 4.3 and 4.4, we propose bespoke languages for cooperative specification (Section 4.5).

Throughout the main sections, we interleave background material and discussion of our findings where it is needed. The remaining sections discuss the relation between our contributions and existing works like programming language features and abstract argumentation frameworks (Section 4.6), lay out the most immediately promising future works (Section 4.8), and reflect on our findings of our contributions and (Section 4.7), before we conclude with a summary (Section 4.9).

## 4.2 Definitions up to Composition Control

This section introduces our fundamental concepts and terminology concerning 1. specification languages, 2. the cooperative specification process, and 3. composition control, which makes a language useful for cooperative specification.

## 4.2.1 Language Schema

We introduce language schemas as abstractions over a class of specification languages. Definition 4.1 gives the formal definition. Intuitively, a language schema defines the (syntactic) program language  $\mathcal{P}$  which agents read, write, compose, and communicate. The model  $\llbracket p \rrbracket$  is the primary (semantic) denotation of each program  $p$  which agents reason about, and use to model the domain of discourse. Secondly, agents recognise when program  $p$  is *valid*, denoted  $\checkmark(p)$ , predicating a fundamental, semantic property like ‘useful’ or ‘sensible’, which is distinct from the model of  $p$ .

**Definition 4.1** (language schema). A language schema is  $\langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$  where:

- $\mathcal{P}$  are the *programs*, a formal language, *e.g.*, defined by a grammar.
- $\mathcal{M}$  are the *models*, another formal language.
- $\llbracket \cdot \rrbracket : \mathcal{P} \rightarrow \mathcal{M}$  maps programs to models. We style the the model of  $p$  as  $\llbracket p \rrbracket$ .
- $\circ : \mathcal{P} \rightarrow \mathcal{P} \rightarrow \mathcal{P}$  composes programs, syntactically.
- $\checkmark : \mathcal{P} \rightarrow \mathbb{B}$  recognises the subset of *valid* programs.

Henceforth, we use *model* to refer specifically to function  $\llbracket \cdot \rrbracket$  and its codomain  $\mathcal{M}$ . We use *semantics* to include any reasoning about programs. We distinguish *static* from *dynamic* semantics: only dynamic semantics is defined in terms of a program’s model. For example, type-checking a program is a matter of static semantics, while testing the model-equivalence of two programs is a matter of dynamic semantics. We use this distinction later to roughly categorise language schemas. For example, Section 4.3.2.4 gives a static definition of *valid*; hence, the validity of these programs can be checked without the cost of computing their models.

**Definition 4.2** (extension).  $\forall p p', (p \circ p')$  *extends*  $p$ .

**Definition 4.3** (adjustment).  $p'$  *adjusts*  $p \triangleq (p' \text{ extends } p) \wedge \checkmark(p) \wedge \checkmark(p')$ .

**Definition 4.4** (refinement).  $p'$  *refines*  $p \triangleq (p' \text{ adjusts } p) \wedge \llbracket p \rrbracket = \llbracket p' \rrbracket$ .

Definitions 4.2 to 4.4 clarify key concepts of our approach. In the context of some language schema  $\langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$ , any program  $p : \mathcal{P}$  can be *extended* to some  $p \circ p' : \mathcal{P}$ . The extension is also an *adjustment* iff it preserves the existing validity of  $p$ . The adjustment is also a *refinement* iff it preserves the model of  $p$ . Intuitively, extensions preserve the existing program syntax, but may change its semantics. Adjustments and refinements are cases of extensions which incrementally preserve more semantic facets of the original program. Although refinements make no immediate change to the program semantics, crucially, they can have an indirect effect, by changing the adjustments and refinements that are available in the future. Example 4.2 demonstrates with a simple and synthetic language schema.

**Example 4.2** (a toy language schema). We demonstrate the use of the language schema with a simple, synthetic example definition of  $\mathcal{S}_{\mathbb{N}} \triangleq \langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$ .

Let the programs ( $\mathcal{P}$ ) be the powerset of natural numbers ( $2^{\mathbb{N}}$ ), *i.e.*, each ‘program’  $p : \mathcal{P}$  is some set of numbers  $p \subseteq \mathbb{N}$ . Let the models ( $\mathcal{M}$ ) be natural numbers, and let  $\llbracket p \rrbracket$  give the maximum of  $p \cup \{0\}$ . Let program composition ( $\circ$ ) be set-union ( $\cup$ ). And finally, let  $\checkmark(p)$  iff no number in  $p$  is divisible by another in  $p$ .

Although artificial,  $\mathcal{S}_{\mathbb{N}}$  demonstrates how we define and reason about language schemas. For example,  $\{5, 6\} \circ p$  is an adjustment if  $p \triangleq \{7\}$ , but not if  $p \triangleq \{3\}$ , because 3 adds a divisor of 6. Furthermore, observe that adjusting  $\{5, 6\}$  with  $\{4\}$  is a refinement, because its model ( $\max(p \cup \{0\}) = 6$ ) and its validity are both preserved. But the refinement has an indirect effect: the refined program  $\{4, 5, 6\}$  has lost a prior adjustment with  $\{8\}$ , because  $\{5, 6, 8\}$  is valid, but  $\{4, 5, 6, 8\}$  is not.

## 4.2.2 Cooperative Specification

Definition 4.5 gives our formalisation of the cooperative specification problem, where a given set of agents cooperate to develop a shared specification.

**Definition 4.5** (cooperative specification). Given language schema  $\langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$ , a set of agents  $A$  take turns developing a persistent and shared program  $p : \mathcal{P}$ . In context, initially  $p$  is either given or trivial. In the  $N^{\text{th}}$  turn, agent  $a \in A$  replaces  $p$  with some extension  $p \circ p'$ . We say  $a$  is the *contributor* of  $p'$ , the  $N^{\text{th}}$  contribution. The cooperation is *successful* while  $p \circ p'$  is an adjustment, *i.e.*,  $p$  remains valid.

We can give an alternative formulation of Definition 4.5 by focusing on the final result of the process: agents cooperate to incrementally develop a shared specification of the form  $((p_1 \circ p_2) \circ p_3) \circ \dots \circ p_n$ , where  $p_N$  is the  $N^{\text{th}}$  contribution. The agents always agree on the fundamental requirements to preserve the success of the cooperation: each contribution must extend the shared specification such that its validity is preserved. Thus, each contribution specifies which contributions are subsequently permitted.

We do not specify how agents take turns, *e.g.*, we generally let agents contribute repeatedly. In practice, we expect the contributor of  $p_N$  to know only the contributions  $p_1, p_2, \dots, p_{N-1}$ , and influence only contributions  $p_{N+1}, p_{N+2}, \dots$  by their choice of  $p_N$ .

**Example 4.3.** Amy and Bob agree to cooperatively define a shared specification, using the toy language schema  $\mathcal{S}_{\mathbb{N}}$ . To the initially trivial  $\{\}$ , Amy contributes  $\{5, 6\}$ , then Bob contributes  $\{13\}$ , and finally Amy contributes  $\{11\}$ . Consequently, Bob and Amy agree on the shared program  $p = \{5, 6, 11, 13\}$ , its model  $\llbracket p \rrbracket = 13$ , and that they cooperated successfully, because the program’s validity was always preserved.

### 4.2.3 Composition Control

We characterise the *composition control* of a given language schema as the set of its *control triples* (Definition 4.6) and use these to reason about the utility of the language. Intuitively, each control triple  $\langle p_1, p_2, p_3 \rangle$  identifies an opportunity to refine  $p_1$  with  $p_2$ , which has no immediate effect on the semantics of  $p_1$ , but it prevents the subsequent adjustment  $p_1 \circ p_3$ . We expect this situation to arise frequently during cooperative specification at the moment an agent is about to update the shared specification to  $p_1$ , but anticipates the adjustment  $p_1 \circ p_3$  by another agent.

In Sections 4.3 and 4.4, we use individual control triples to concretise cases where agents control one another’s contributions, and we reason about control triples in aggregate as a ‘measure’ for the composition control of a language (schema).

**Definition 4.6** (control triples).  $\langle p_1, p_2, p_3 \rangle \in \text{control-triples}(\langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle) \triangleq (p_2 \text{ refines } p_1) \wedge (p_3 \text{ adjusts } p_1) \wedge \neg(p_3 \text{ adjusts } (p_1 \circ p_2)).$

**Example 4.4.** Language schema  $\mathcal{S}_{\mathbb{N}}$  affords non-trivial composition control. In Example 4.3, the agents’ contributions align with non-trivial control triples in  $\mathcal{S}_{\mathbb{N}}$ , *i.e.*, their contributions non-trivially specify their peers’ permitted contributions.

Amy’s first contribution  $\{5, 6\}$  prohibited the subsequent contribution of  $\{10\}$ . Amy’s inclusion of 5 in the contribution had no immediate effect on the model of the shared specification, but it constrained future contributions. This situation aligns with the control triple  $\langle \{6\}, \{5\}, \{10\} \rangle$  as  $\llbracket \{6, 10\} \rrbracket = \llbracket \{5, 6, 10\} \rrbracket = 10$  and  $\checkmark(\{6, 10\})$  but  $\neg\checkmark(\{5, 6, 10\})$ . Amy’s prohibition of Bob’s contribution of  $\{10\}$  was evident to Bob, but Amy’s underlying motivation was not. Did Amy consider 10 to be undesirable?

## 4.3 Composition Control Features in Datalog

This section presents a set of generic, composition control language features, using Datalog as a specification language in each example. We select Datalog in particular, as it strikes a desirable compromise between complexity (affording varied and meaningful specifications) and simplicity (affording succinct definitions and examples).

### 4.3.1 The Datalog Language

Datalog has been discussed in the literature for decades, including a rich exploration of various language extensions such as *weak negation* and *composite objects*; [CGT89] overviews the Datalog language variants, but the original Datalog language is sufficient for our purposes. We assume some prior knowledge on Datalog but give our own semi-formal account of the language here.

Definition 4.7 gives our account of the syntax of the Datalog language.

As in the subsequent grammars, note how grammar non-terminal symbols are suggestively named, like *rule* and *predicate*. Thus, our grammars introduce abstract language constructs. However, the grammar rules ultimately terminate with concrete ASCII symbols. Thus, we rely on our grammars to support concrete examples.

**Notation 4.1** (formal language grammars). Throughout this chapter, we define several formal languages inductively, via collections of grammar rules. Syntactic categories are defined by non-terminal symbols, denoted in italics (like *rule*). We omit the definitions of common and trivial syntactic categories and grey their symbols (like *letter*). Concrete, terminal tokens are mono-spaced and brightly coloured (like `statement` and `:-`). Each `:=` defines a grammar rule, rewriting from left to right, with alternatives separated by `|`. We use  $x?$  to denote an optional  $x$  element and  $\epsilon$  to mark the absence of tokens in rules rewriting non-terminal symbols to nothing. We use  $x^*$  to denote  $x$ -lists. Precisely, let  $x^* \triangleq \epsilon \mid x x^*$ . Finally,  $x^{y+}$  and  $x^{y*}$  match lists with at least 1 and 0 elements, respectively, whose elements are separated by elements matching  $y$ . Precisely, let  $x^{y+} \triangleq x (y x)^*$  and let  $x^{y*} \triangleq x^{y+}?$ . For example, the one-rule grammar ( $D := -? \textit{digit}^+$ ) lets  $D$  denote the signed integers such as `-23`.

**Definition 4.7** (Datalog syntax). Let *program* give the Datalog programs, where

<i>program</i> := <i>rule</i> <sup>*</sup>	<i>rule</i> := <i>head</i> ( <code>:-</code> <i>body</i> )?	<i>predicate</i> := <i>constant</i>
<i>head</i> := <i>atom</i>	<i>atom</i> := <i>predicate</i> ( <code>(</code> <i>arg</i> <sup>*</sup> <code>)</code> )?	<i>variable</i> := <i>uppercase letter</i> <sup>*</sup>
<i>body</i> := <i>atom</i> <sup>*</sup>	<i>arg</i> := <i>variable</i>   <i>constant</i>	<i>constant</i> := <i>lowercase letter</i> <sup>*</sup>

**Definition 4.8** (ground atom). Each atom  $\alpha$  is *ground* iff it contains no variable.

Datalog affords a straightforward operational semantics: rules populate an initially-empty set of *true ground atoms* (Definition 4.8) from existing truths to a fixed point. Desirably, each Datalog program also affords a straightforward logical interpretation: rules are implications, bodies are conjunctions of atoms, and atoms are first-order, atomic, Boolean logical variables. For instance, Example 4.5 has the model with truths `{ no-access(amy,data1) , access(amy,data1) , bad(amy) }`. Each model assigns a Boolean truth value to every conceivable ground atom, thus determining the Boolean value of every conceivable query over their values.<sup>1</sup> For example, `bad(amy)` is true, while `bad(bob)` and `eats(bob,baked,potatoes,500)` are each false.

**Example 4.5** (a simple Datalog program).

```
bad(A) :- no-access(A,D), access(A,D).    no-access(amy,data1).    access(amy,data1).
```

Despite its simplicity, Datalog has seen significant application in practice, because it strikes a desirable balance of expressivity, *e.g.*, it is simple enough to afford efficient interpretation, but complex enough to afford complex modelling and analyses.

<sup>1</sup>Syntactically, Clingo is a super-language of Datalog, so the Clingo solver accepts Datalog programs as input. By design, these inputs yield unique stable models which align with the semantics of Datalog. So the reader can apply the understanding of Clingo from Section 2.3 here.

## 4.3.2 Composition Control in Datalog Variants

### 4.3.2.1 Datalog with Trivial Validity

Definition 4.9 defines  $\mathcal{S}_D$ , which situates the Datalog language in a simple language schema. Consequently, agents can cooperatively specify Datalog programs.

**Definition 4.9** (trivially valid Datalog). Let  $\mathcal{S}_D \triangleq \langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$ , where:

- $\mathcal{P} \triangleq \text{program}$ , the syntactic category of Datalog programs (Definition 4.7).
- $\mathcal{M}$  is the powerset of ground atoms (Definition 4.8), *i.e.*, models are atom-sets.
- $\llbracket \cdot \rrbracket$  captures the typical Datalog semantics.
- $\circ$  is given by the usual, syntactic program concatenation, such that the composite program  $p_1 \circ p_2$  has the rules of the union of the rules in programs  $p_1$  and  $p_2$ .
- $\checkmark$  is a trivial, constant function: each program is valid.

$\mathcal{S}_D$  has some properties that support its usage for cooperative specification. Notably, it reflects a hallmark of many Datalog variants in the literature: the semantics disregards the order of program rules. Precisely, in  $\mathcal{S}_D$ ,  $\circ$  associates and commutes over  $\llbracket \cdot \rrbracket$ . In our context, this eases the burden on agents cooperating to define any  $p_1 \circ p_2$ , because there is no ordering imposed on their contributions.

However,  $\mathcal{S}_D$  is not very useful for cooperative specification; it has no meaningful composition control, because there are no control triples, as all programs are valid. Example 4.6 demonstrates how the lack of composition control is experienced as an inability for the agents to communicate their requirements on their cooperation.

**Example 4.6** (trivial Datalog cooperation). Agents Amy and Bob cooperate to define composite  $p_1 \circ p_2$ . By defining  $p_1$ , Amy constrains  $\llbracket p_1 \circ p_2 \rrbracket$ , but not  $p_2$ .

For example, if  $p_1 \triangleq \text{no-access}(\text{amy}, \text{data1})$ , Amy asserts that  $\text{no-access}(\text{amy}, \text{data1}) \in \llbracket p_1 \circ p_2 \rrbracket$ . However, Amy cannot control, and Bob cannot infer, which definitions of  $p_2$  Amy considers permissible. Did Amy omit  $\text{no-access}(\text{bob}, \text{data1})$  intentionally? Is Amy leaving this addition to Bob, or does Amy not permit its truth?  $\mathcal{S}_D$  programs do not specify how they may be changed. Amy experiences this as a lack of control over Bob. Bob experiences this as a lack of insight into what Amy permits.

### 4.3.2.2 Simple Dynamic Validity

Next, Definition 4.10 situates the same Datalog language in a new language schema. We attribute semantic invalidity to the truth of **error**, which is otherwise an entirely ordinary truth of an entirely ordinary atom. Despite the simplicity of this definition, it makes  $\mathcal{S}'_D$  significantly more useful for cooperative specification than  $\mathcal{S}_D$ . Example 4.7

discusses an control triple which is illustrative of the practical usage of  $\mathcal{S}'_D$ , and which witnesses its non-trivial composition control.

**Definition 4.10** (error-sensitive Datalog). Let  $\mathcal{S}'_D \triangleq \mathcal{S}_D$ , except  $\checkmark(p) \triangleq \text{error} \notin \llbracket p \rrbracket$ .

**Example 4.7** ( $\mathcal{S}'_D$  composition control).  $\langle p_1, p_2, p_3 \rangle \in \text{control-triples}(\mathcal{S}'_D)$ , where

$$p_1 \triangleq \text{knows}(\text{amy}, \text{bob}). \quad p_2 \triangleq \text{error} :- \text{knows}(\text{X}, \text{dan}). \quad p_3 \triangleq \text{knows}(\text{amy}, \text{dan}).$$

Extrapolating from Example 4.7,  $\mathcal{S}'_D$  affords a straightforward means of composition control: agents express a prohibition on (contributions that lead to) the simultaneous truth of  $a_1, a_2, a_3, \dots, a_n$  as a rule  $\text{error} :- a_1, a_2, a_3, \dots, a_n$ .<sup>2</sup>

We call this a *dynamic* form of composition control, as it entangles each program's (in)validity with its model. Intuitively,  $\mathcal{S}'_D$  frames Datalog as a (simple) *homogeneous meta-programming* language: Datalog constructs specify the use of Datalog constructs. On the one hand, this approach has the drawback of complexity; generally, checking (in)validity of any program  $p$  requires computing  $\llbracket p \rrbracket$  in its entirety. On the other hand, this approach has the benefit of expressivity; as demonstrated in Example 4.7, it lets ordinary Datalog rules express concerns that cut across the traditional separation between domain-level concepts (like the `knows` predicate) and the meta-level (like constraints on contributions). Example 4.8 recreates Example 4.6, but where the cooperation is more fruitful, because Amy is able to explicitly express more meta-level requirements to Bob. However, note that the requirement is not specific to Bob:  $\mathcal{S}'_D$  cannot specialise the conditions on contributions according to their contributor.

**Example 4.8** (prohibiting no-access). As in Example 4.6, Amy and Bob cooperate to develop the shared specification  $p_1 \circ p_2$ , but this time, they agree to use  $\mathcal{S}'_D$ . Amy defines  $p_1 \triangleq \text{no-access}(\text{amy}, \text{data1})$ .  $\text{error} :- \text{no-access}(\text{bob}, \text{data1})$ , and leaves  $p_2$  for Bob to define. It is clear to Bob that the success of the cooperation is conditioned on  $p_2$  preserving the falsity of `no-access(bob, data1)`. Amy experiences this as control over Bob. Bob experiences this as insight into what Amy permits. As in Example 4.6,  $\text{no-access}(\text{bob}, \text{data1}) \notin \llbracket p_1 \rrbracket$ , but now this was intended by Amy.

### 4.3.2.3 Dynamic Validity Reflecting Contributors

Recall how cooperative specification (Definition 4.5) statically constrains the ways agents modify the shared specification, establishing new meaningful abstractions of *contribution* and *permission*, to guide the agents' cooperation. Here, we introduce another static constraint which establishes a connection between contributions and

<sup>2</sup>The reader may recognise how the rules of this form mimic the *integrity constraint rules* used in answer-set programming languages such as Clingo (see Section 2.3). The rules have the same structure, and assign a similar interpretation of 'to be avoided' to the applicability of these constraint rules. However, by separating the semantic facets of programs into models ( $\llbracket \cdot \rrbracket$ ) and validity ( $\checkmark$ ), we let users encounter, reason about, and even contribute programs that violate constraints.

their contributors, such that specifications of the latter can be specialised to the former: Property 4.1 characterise contributions that *reflect their contributors*.

**Property 4.1** (contributions reflect contributors). Let language  $\mathcal{S} \triangleq \langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$  be given. The contribution of  $p \in \mathcal{P}$  by agent  $a$  is permitted only if  $p$  *reflects* or *is consistent with* being contributed by  $a$ : no agent other than  $a$  apparently contributes to  $p$ , where *apparently contributes to* is defined alongside  $\mathcal{S}$ .

By agreeing *a priori* that contributions must reflect contributors (Property 4.1), contributors are forced to internalise details of their cooperation in the specification itself. Intuitively, contributors can exploit the required connection between contributors and contributions to specify who contributes what. For example, after agents agree on Definition 4.11,  $\mathcal{S}'_D$  support new usages. Example 4.9 demonstrates; Amy exploits the connection between Bob’s contributions and Bob’s identity.

**Definition 4.11** (apparently contributes). Agent  $a_1$  *apparently contributes* (to predicate  $c$ ) in Datalog program  $p$  iff  $p$  contains a rule with head  $c(a_1, a_2, a_3, \dots, a_n)$ .

**Example 4.9** (agent-specific contribution constraints). Like Example 4.8, Amy and Bob cooperate to develop a shared specification after agreeing to use  $\mathcal{S}'_D$ , but now, they also agree that contributions must *reflect* their contributors (Property 4.1) in the first argument to each contributed rule’s head (Definition 4.11).

First, Amy contributes `votes-for(amy, bob). error :- votes-for(x, x)`. As this prohibits the inference of `votes-for(dan, dan)`, *only* Dan is not permitted to vote for Dan.

On the one hand, the preservation of Property 4.1 has the drawback of complicating specifications and models, because it forces contributors to work around the added constraints. For example, the `no-access` predicate used in Example 4.9 would require an additional parameter to identify the contributor distinctly from the data-accessor. On the other hand, it connects another facet of the cooperation at the meta-level with the models at the domain-level, affording new kinds of inter-agent control. For example, in all successful contributions which continue Example 4.8, we know that agents cannot vote for themselves, so Dan cannot acquire votes without other agents’ assistance. In Chapter 6, we extensively apply these agent-specific constraints.

#### 4.3.2.4 Static Validity via Type-Sealing

Finally, Definition 4.15 defines  $\mathcal{S}''_D$  as another language schema, framing Datalog for cooperative specification. Like  $\mathcal{S}'_D$ , this is a variant of the original schema for Datalog  $\mathcal{S}_D$ , but unlike  $\mathcal{S}'_D$ , this is designed around a *static* approach to composition control: checking the (in)validity of each program is orthogonalised from its model. As before, agents’ contributions must reflect their contributors (Property 4.1) in the first parameter of the head of each contributed rule (Definition 4.11).

**Definition 4.12** (extended Datalog syntax). Let  $program'$  give the syntactic category of extended Datalog programs  $program'$ , subsuming  $program$  from Definition 4.7:

$$\begin{aligned} program' &:= phrase^* & agent &:= constant \\ phrase &:= \text{rule rule} \mid \text{seal predicate except by agent}^+ \end{aligned}$$

**Definition 4.13** (sealed for). In each  $p \in program'$ , predicate  $c$  is sealed for agent  $a$  iff  $\exists(\text{seal } c \text{ except by } a_1, a_2, a_3, \dots, a_n) \in p$  and  $a \notin \{a_1, a_2, a_3, \dots, a_n\}$ .

**Definition 4.14** (breaks seal). Program  $p \circ p'$  breaks a seal on predicate  $c$  iff  $c$  is sealed for some agent  $a$  in  $p$  and  $a$  apparently contributes to  $c$  in  $p'$ .

**Definition 4.15** (sealing Datalog). Let  $S_D'' \triangleq S_D$ , except defined with  $\mathcal{P} \triangleq program'$  (Definition 4.12), and  $\checkmark(p)$  iff no seal is broken in  $p$  (Definition 4.14).

The idea of *seals* is straightforward (and is adapted from the SEASO language in Chapter 3), but it requires a syntactic extension to Datalog in Definition 4.12: the existing rules are generalised to *phrases*, whose first case of *rule*-phrases express logical inference rules as before. The new case of *seal*-phrases are used to express constraints on contributions at the meta-level. Intuitively, seal-phrases ('seals') prohibit certain contributions in the future, relying on the alignment between the (syntactic) order of phrases in the program with the (chronological) order of contributions.

**Example 4.10** (simple control via sealing). Amy and Bob cooperate to develop a shared specification using  $S_D''$ . First, Amy contributes `seal ready except by amy`. Everyone agrees that Amy, but not Bob, is not permitted to assert that `ready` is true.

**Example 4.11** (subtle control via sealing). Amy and Bob continue the cooperation in Example 4.10. Amy contributes `ready :- vote(bob)`. Bob then contributes `vote(bob)`, indirectly making `ready` true. Everyone agrees that this is permitted: Bob has not apparently contributed to `ready`, and all contributions still reflect their contributors.

Compared to the dynamic validity of  $S_D'$ , the static validity of  $S_D''$  has some drawbacks. Notably, by separating the reasoning about (fine-grained) truths in the model from the (coarse-grained) rules in the program, fine-grained meta-level requirements become more cumbersome or even impossible to express. For example, in any cooperation, Amy cannot statically prohibit Bob from asserting `vote(amy)` but not `vote(bob)` without at least partially entangling (the computation of) the program's model with its validity. However, there are also benefits to static validity: it affords more static (analysis of) properties. For example, for each control triple  $\langle p_1, p_2, p_3 \rangle$  of  $S_D''$ , necessarily a seal in  $p_2$  is broken by a rule in  $p_3$ , or *vice versa*. This lets agents use seals to reason about valid programs in terms of their parts. For example, given  $p_1 \triangleq (\text{no-access}(\text{bob}, \text{bob}, \text{X}) \text{ :- private}(\text{X}). \text{seal no-access except by bob})$ , for any  $p_2$ , necessarily  $\checkmark(p_1 \circ p_2) \rightarrow \text{no-access}(\text{amy}, \text{data1}) \notin \llbracket p_1 \circ p_2 \rrbracket$ . Agents can draw this sort of conclusion even if they have incomplete knowledge of the shared program.

## 4.4 Adapting Existing Languages for Cooperative Specification

This section adapts the approaches to improving composition control from Section 4.3 to realistic specification languages Alloy (Section 4.4.2) and eFLINT (Section 4.4.3).

For each language, we first give an account of the language, simplified for our purposes (as we detail in a dedicated paragraph), and we remark on its current use for (data exchange) system specification. Then, we demonstrate and evaluate its use for cooperative specification. Finally, we define a minor language extension, and then demonstrate and evaluate the improvements to its composition control.

### 4.4.1 Running Example Usage Scenario

Here, we present a particular cooperative specification scenario to be used throughout this section. This makes the demonstrations more predictable to the reader, and affords some comparison between the findings of Alloy and eFLINT.

The following three agents cooperate in the development of a shared specification. Each agent is concerned with their own facet of the system and their own requirements:

- **‘Administrator’ Amy** is concerned with controlling the distributed infrastructure, for example, by defining processing events, and specifying liveness properties.
- **‘Barrister’ Bob** is a legal expert, and is concerned with enforcing an interpretation of a core part of the GDPR [Eur16]: the lawfulness of data-processing under GDPR Article 6(1)(a) requires the consent of the subject of the processed data.
- **‘Data-processor’ Dan** is a user of the infrastructure, and is concerned with driving the development of the specification to reflect Dan’s processing of data.

To simplify our examples (and focus on matters of formal languages, syntax, semantics, etc.), our interpretations of the GDPR are simplistic. Notably, we often obligate Dan as the *data processor* to acquire consent to process the data, which is normally the legal obligation of the *data controller*, which may be distinct entity.

### 4.4.2 Cooperative Specification in Alloy

#### 4.4.2.1 An Account of Alloy

Alloy has seen significant research and application to the modelling and model-checking of software systems [Jac03]. Reflecting its basis on UML and OML [He06], Alloy specifications represent data via named relations and constraints over relations expressed in a first-order logic based on Tarski’s calculus of relations [Tar41]. Alloy has seen application to DSL engineering [MP15] and for the static analysis of complex

systems [Jac19]. The Alloy website<sup>3</sup> provides release versions of Alloy tools, tutorials, Alloy reference examples, and a detailed specification of the Alloy syntax.

The meaning of each Alloy specification  $p$  is a set of *instances* which *satisfy*  $p$ . Each instance prescribes particular members to named  $N$ -ary relations. Each instance formalises a satisfactory system configuration, such that Alloy specifications predicate satisfactory configurations. The names and types of the relations are determined by each *signature* in  $p$ , which defines an *atomic* relation  $r$ , and several *fields* over  $r$  and other atomic relations. Here, Alloy affords ease of use by leveraging the expected familiarity of its users with the object-oriented paradigm: each signature corresponds to a record data type. Thus, signatures specify the structure of the elements in instances. *Facts* explicitly constrain instances in terms of *predicates*, first-order logical formulae over relations and their members. Given a specification, the *AlloyAnalyzer* tool reports the satisfying instances. In practice, they are enumerated one at a time, at the request of the user to proceed. This accounts for the possibility that there are more satisfying instances than can be enumerated in practice. Sometimes it suffices to check if the specification's satisfiability is witnessed by any instance at all.

Definitions 4.16 and 4.17 give our syntactic Alloy programs and semantic Alloy models, *i.e.*, the input and output languages of the AlloyAnalyzer.

**Definition 4.16** (Alloy syntax). Let *program* define the (syntactic) language of Alloy specifications, the input language of the AlloyAnalyzer tool.

$$\begin{aligned}
 \text{program} &:= \text{para}^* & \text{para} &:= \text{macro} \mid \text{factDef} \mid \text{sigDef} \\
 \text{id} &:= \text{letter}^+ & \text{factDef} &:= \text{fact id? } \{ \text{pred} \} \\
 \text{fieldDef} &:= \text{id} : (\text{mult} \mid \text{set})? \text{rel} & \text{macro} &:= \text{let id} = (\text{pred} \mid \text{rel}) \\
 \text{mult} &:= \text{one} \mid \text{none} \mid \text{some} & \text{sigDef} &:= \text{mult? sig id}^+ (\text{in rel})? \{ \text{fieldDef}^* \} \\
 \text{bin} &:= . \mid \& \mid \rightarrow \mid + & \text{rel} &:= \text{id} \mid \text{none} \mid ( \text{rel} ) \mid * \text{rel} \mid \text{rel bin rel} \\
 \text{pred} &:= \text{id} \mid \text{true} \mid ( \text{pred} ) \mid \text{not pred} \mid \text{pred and pred} \\
 & & & \mid \text{all id} : \text{rel } \{ \text{pred} \} \mid \text{some rel} \mid \text{rel} (= \mid \text{in}) \text{rel}
 \end{aligned}$$

**Definition 4.17** (Alloy models). *instanceSet* defines the (semantic) language of Alloy instance-sets, the output language of the AlloyAnalyzer tool.

$$\begin{aligned}
 \text{instanceSet} &:= \text{instance}^* & \text{instance} &:= \{ \text{atom} \mid \text{field} \} \\
 \text{field} &:= \text{id} ( \text{atom}^* ) & \text{atom} &:= \text{id digit}^+
 \end{aligned}$$

**Example 4.12** (a simple Alloy example).

```
sig Agent, Data {}
let Processor = Agent
```

<sup>3</sup><https://alloytools.org/>

```

sig State { step: set State, process: set      Data -> Processor ,
           consent: set Agent -> Data -> Processor }
fact allReachableStep { all s:State { s.*step = State }}
fact processingImpliesConsent { all s:State { s.process in Agent.(s.consent) }}

```

Run with this specification  $\in$  *program* as input, the AlloyAnalyzer outputs satisfying instances  $\subseteq$  *instanceSet*:

```

{ },
{ Agent1 },
{ Agent1, Agent2 },
...
{ State1, step(State1, State1), Agent1, Agent2, Data1, consent(Agent1, Data1, Agent1) },
...

```

Example 4.12 illustrates Alloy in action. The first paragraph of the specification (in *para*) introduces atomic relations `Agent` and `Data`. The second paragraph defines `Processor` as an alias of `Agent`; this affords more suggestive human interpretation of the field-relations over agents playing multiple roles. The third paragraph introduces the final atomic relation, `State`, along with three field relations. Here, `->` (set product, conventionally notated  $\times$ ) is used to construct new set-expressions, and `set` is used to remove the default multiplicity constraint (`one`: exactly one element) on field relations; *e.g.*, such that `step` is any subset of `State -> State`. Note that each field-relation implicitly includes the signature-type itself as the first parameter. For example, each `process` member is an state-data-agent triple. The final paragraphs define facts that explicitly constrain the satisfying instances. For example, `allReachableStep` asserts that each states reaches each state via steps ( $R^*$  is the transitive closure of  $R$  and  $(=)$  asserts extensional set equality). Example 4.12 shows four satisfying instances.

**Simplifications** Our simplified Alloy language is a sub-language of the real Alloy, so it is executable using the real AlloyAnalyzer. We make some simplifying omissions:

- We omit many combinators for predicates (like `iff`) and relations (like `>`).
- To simplify the (user control over) scope of instance enumeration, we omit paragraphs enabling fine control over the AlloyAnalyzer’s search for satisfying instances. In real Alloy, it is common practice to interleave facts with `run` and `check` commands, which guide the search for instances. We omit the constructs for overriding the default *search scope*: the cardinality of atomic relations.
- We omit the language features added in Alloy version 6, including temporal operators (like `until` and `before`), and generalisation of instances to linear traces. Alloy version 6 is backwards-compatible, and its novelties can be modelled using standard Alloy features<sup>4</sup> We take this approach in our Alloy examples.

<sup>4</sup>In the official Alloy tutorial, the *river crossing* example models states and temporal properties using the standard relations and facts <https://alloytools.org/tutorials/online/frame-RC-1.html>.

#### 4.4.2.2 Simple Composition Control in Alloy

Definition 4.18 embeds Alloy straightforwardly in language schema  $\mathcal{S}_A$ , enabling its application to cooperative specification. Until Section 4.4.2.3, we drop the requirement that contributions reflect their contributors (Property 4.1), because it requires adapting the definition of *apparently contributes* from Datalog (Definition 4.11).

**Definition 4.18** (satisfiable Alloy). Let  $\mathcal{S}_A \triangleq \langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$ , where:

- $\mathcal{P} \triangleq$  *program* (Definition 4.16), the syntactic Alloy specification language.
- $\mathcal{M} \triangleq$  *instanceSet* (Definition 4.17), the powerset of Alloy instances.
- $\llbracket \cdot \rrbracket$  captures the dynamic semantics of Alloy: enumerating the satisfying instances.
- $\circ$  concatenates programs in  $\mathcal{P}$ .
- $\checkmark(p)$  recognises when 1.  $p$  satisfies the static semantics of Alloy: identifiers are bound by quantifiers or resolve to either identifier- or predicate-definitions, and expressions are well-typed, and 2.  $\llbracket p \rrbracket \neq \{\}$ , *i.e.*,  $p$  is satisfiable.

$\mathcal{S}_A$  faithfully captures the characteristic declarative nature of Alloy specifications; notably, paragraphs in  $p$  interact only indirectly, via the instances  $\llbracket p \rrbracket$ . Precisely, adding signatures multiplies the elements of  $\llbracket p \rrbracket$ , and enables the subsequent addition of new signatures and facts. Adding facts to  $p$  removes elements from  $\llbracket p \rrbracket$ . The specification is invalidated once the last satisfying instance is removed.

$\mathcal{S}_A$  exhibits considerable composition control. It has control triples of the form  $\langle p_1, p_2, p_3 \rangle$  such that  $\llbracket p_1 \circ p_2 \rrbracket$  and  $\llbracket p_1 \circ p_3 \rrbracket$  are non-empty, but  $\llbracket p_1 \circ p_2 \circ p_3 \rrbracket$  is empty. Generally, facts constrain the satisfying instances independently. For example, if  $p_2$  and  $p_3$  contain only facts, then  $\llbracket p_1 \circ p_2 \circ p_3 \rrbracket = \llbracket p_1 \circ p_2 \rrbracket \cap \llbracket p_1 \circ p_3 \rrbracket$ .  $\mathcal{S}_A$  has some use in cooperative specification, because facts afford composition control. However, invalidity is witnessed by no instances, so causes of invalidity are difficult to diagnose.

**Example 4.13** (successful cooperation until overconstraint). Agents Amy, Bob, and Dan cooperate to develop a shared specification. Amy contributes first, capturing the fundamental dynamics of data-processing as a *state-step* transition system, capturing processing and consent as changing relations over agents and data.

```
sig Agent, Data {}      let Processor = Agent
sig State { step: set State, process: set      Data -> Processor ,
              consent: set Agent -> Data -> Processor }
```

Next, Bob captures a legal requirement: each data has exactly one subject. Moreover, Bob formalises the role of `consent`: only a data subject can consent to its processing. To add new relations over existing `Data`, Bob uses a simple Alloy encoding trick: `▷` is introduced with novel fields, and then the `▷` and `Data` relations are unified.

Next, Amy specifies a basic liveness property: from any state, there exists a sequence of steps that result in the creation of any conceivable consent.

```
fact anyConsentAlwaysReachable {
  all s: State { s.*step.consent = Processor -> Data -> Agent }}
```

Finally, Dan considers making the following modest contribution. It specifies that Dan and Amy are distinct agents, and that Dan processes some data. However, Dan recognises that it is not permitted, because it invalidates the shared specification. Unfortunately, the reason why is not clear. The AlloyAnalyzer output of {} does not help to clarify. Although the agents maintain agreement on the progress and success of the cooperation, its productivity breaks down as Dan struggles to predict, understand, and explain which contributions are permitted.

```
one sig Amy, Dan in Agent {} fact { not Amy = Dan }
fact danProcesses { some State.process.Dan }
```

Example 4.13 demonstrates how  $\mathcal{S}_A$  enables productive cooperative specification to some extent, but problems arise in practice. In this case, Dan encountered an unexpected prohibition. It was unexpected because it emerged from the complex interactions between prior contributions. In this case, the underlying cause was a subtle conflict between the roles Amy and Bob impose on `consent` as captured in `anyConsentAlwaysReachable` and `onlySubjectsConsent`, respectively. Together, these facts imply that either `State` is empty, or `Agent` is a singleton. Amy and Bob cooperated ineffectively, but this was not captured by the specification. In general, like  $\mathcal{S}'_D$ ,  $\mathcal{S}_A$  exhibits powerful and complex dynamic composition control. However, invalidity in  $\mathcal{S}_A$  is very difficult to diagnose, because unlike the inference of the Datalog atom `error`, the models of unsatisfiable Alloy specifications afford no starting point for explanation or diagnosis; there simply are no instances! In Example 4.13, we can frame the problem as Bob's inability to overrule Amy in the legal matter of consent.

#### 4.4.2.3 Improved Composition Control in Alloy

Definition 4.21 defines  $\mathcal{S}'_A$  as a variant of  $\mathcal{S}_A$  in the same way that  $\mathcal{S}'_D$  was defined as a variant of  $\mathcal{S}_D$  in Section 4.3.2.4: the syntax of specifications is extended with new *seal* constructs, and the semantics of validity is changed to make seals impose static constraints on future contributions as a function of the apparent contributors to their predicates. As such, we reintroduce the requirement that contributions reflect their contributors (Property 4.1). This depends on a definition of *apparent contributors* to (extended) Alloy specifications in Definition 4.27, which adapts the original Definition 4.11 from the Datalog language to our simplified Alloy language.

**Definition 4.19** (extended Alloy syntax). Let  $program'$  give the syntactic category of syntactically-extended Alloy programs (extending Definition 4.16), where:

$$\begin{aligned} program' &:= paraExt^* & agent &:= id \\ paraExt &:= agent \text{ states } para \mid seal \text{ id } \text{except by } agent \end{aligned}$$

**Definition 4.20** (apparent paragraph contributor). Agent  $a$  *apparently contributes* to  $(i : id \text{ in } p : program' \text{ iff } \exists x : para, (a \text{ states } x) \in p$  and  $i$  occurs within  $x$ .

**Definition 4.21** (improved Alloy). Let  $S'_A \triangleq S_A$ , except defined with  $\mathcal{P} \triangleq program'$  (Definition 4.19), and  $\checkmark(p)$  iff no seal is broken in  $p$  (Definition 4.14).

$S'_A$  includes the control triples of  $S_A$ . Moreover,  $S'_A$  has new control triples matching  $\langle p_1, p_2, p_3 \rangle$  in which a seal in  $p_2$  is broken in  $p_3$  or *vice versa*. Broken seals are discovered without the need to compute models, and their breakage can be explained to users in straightforward, syntactic terms.

In general,  $S'_A$  programs are still subject to (unintentional) invalidity via overconstraint that is difficult to diagnose and correct. However, when contributors anticipate undesirable contributions, these can be explicitly internalised in the specification via seals. In these cases, seals explicitly capture and communicate these constraints in a form that is easy to understand and diagnose.

**Example 4.14** (Alloy cooperation controlled by sealing). We adapt the cooperation between Amy, Bob, and Dan in Example 4.13. The contents of each contribution is largely unchanged, except that each rule is prefixed by  $(\alpha \text{ states})$  such that it reflects the identity of its contributor:  $\alpha$ . Also, Bob's contribution includes a new extended statement: `seal consent except by bob`, to prevent other contributions from making inappropriate use of the `consent` relation (intentionally or otherwise). Consequently, Amy finds that `anyConsentAlwaysReachable` cannot be contributed as intended, as doing so would break Bob's seal on `consent`. Bob experiences this as control over Amy. Amy experiences this as insight into which contributions Bob considers permissible.

### 4.4.3 Cooperative Specification in eFLINT

Here, we explore the application of eFLINT to cooperative specification. Our contribution here is independent of that of Chapter 2, where we re-design eFLINT. So what we call eFLINT here is called the *original eFLINT* in Sections 2.1 and 2.2.

#### 4.4.3.1 An Account of eFLINT

eFLINT is a domain-specific specification language suited to formalising a variety of sources of norms [vBLvDvE20]. It sees active development and application to the regulation of cyber-physical systems for which compliance to regulatory norms is

important, e.g., medical data processing systems. As such, the design of eFLINT reflects an emphasis on modelling via abstractions that connect normative concepts (like actions and duties) and computer systems (like state transitions). The language emphasises the extensibility of specifications to reflect the dynamism of norms in practice, for example, to mirror amendments to external legal regulations [vBKB<sup>+</sup>21].

An eFLINT specification models a snapshot of a cyber-social system as a finite collection of typed *instances*<sup>5</sup>, which are separated into two categories:

1. The set of *t*-type instances that *hold* denote the elements of the relation corresponding to *t*. These instances build up the model of the domain of discourse, modelling anything from metadata to normative relations between entities.
2. The *violations* are instances that witness the deviations from the specification. They model anything undesirable as unmet obligations or prohibited actions.

eFLINT reflects an emphasis on *dynamism* at several levels. Firstly, the language includes constructs that let statements appended to the end of the specification modify the type-definitions and instances at the domain level. This reflects eFLINT's use as in input language to an *incremental* interpreter: statements added to the specification later can act as refinements to prior definitions, e.g., mimicking the way legal documents are continuously amended. Secondly, statements are explicitly grouped into *steps*, where each new step advances a conceptual clock, transitioning the system being modelled into a new state. For example, the eFLINT interpreter can act as a monitor: each input step updates its view of a changing system.

Definitions 4.22 and 4.23 give our syntactic eFLINT programs and semantic eFLINT models, *i.e.*, the input and output languages of the eFLINT interpreter.

**Definition 4.22** (eFLINT syntax). Let *program* define the (syntactic) language of eFLINT specifications, the input language of the eFLINT interpreter tool.

$program := step^*$	$alias := Placeholder\ type\ For\ type$
$step := stmt^+ .$	$type := lowercase(-   _   letter)^*$
$var := type (digit^*   ' ^*)$	$fDef := (Fact\ type\ fSig   Extend\ Fact\ type)\ clause^*$
$stmt := fDef   dDef   alias$	$dDef := (Duty\ type\ dSig   Extend\ Duty\ type)\ dClause^*$
$fSig := Identified\ by\ var^{*+}$	$dSig := Holder\ var\ Claimant\ var\ (Related\ to\ var^{*+})?$

<sup>5</sup>Note that both Alloy and eFLINT use the term *instance*, but very differently: eFLINT instances are analogous to what are called *atoms* in Datalog and Alloy.

```

clause := Derived from instExpr'* | Conditioned by boolExpr
dClause := Violated when boolExpr | clause
instExpr := string | digit+ | instExpr (+ | - | *) instExpr | var | instExpr . var
           | struct | ( ForEach var : instExpr ) | instExpr Where boolExpr
struct := type ( instExpr'* )
boolExpr := True | Not ( boolExpr ) | Holds ( instExpr ) | instExpr (!= | == | <=) instExpr

```

**Definition 4.23** (eFLINT models). Let *report* define the semantic language of eFLINT specifications, the output language of the eFLINT interpreter tool.

```

report := state*           state := hold: instSet  violate: instSet
instSet := { inst'* }     inst := string | type ( inst'* )

```

**Example 4.15** (model of a simple eFLINT specification).

```

Fact agent Identified by string
Fact data  Identified by string Derived from data("CatScans"), data("CatScans")
.
Extend Fact data Derived from data("X-Rays")
.
Fact data    Identified by subject * string
.
Placeholder subject For agent
Placeholder processor For agent
Fact process Identified by processor * data
Fact consented Identified by process
Fact data    Identified by subject * string
Duty getConsent Holder processor Claimant subject Related to process
    Violated when Not(consented(process))
    Derived from (ForEach process:
        getConsent(process.processor, process.data.subject, process))
.
Extend process Derived from process(processor("Amy"),data(subject("Bob"),"X-Rays"))
.
Extend Fact process Conditioned by processor != subject("Amy")
.

```

Run with the above eFLINT specification  $\in$  *program* as input, the eFLINT reasoner outputs the model  $\in$  *report* below.

```

hold: {} violate {}.
hold: { data(string("CatScans")), } violate: {}.
hold: { data(string("CTScans"), data(string("X-Rays"))) } violate: {}.
hold: {} violate {}.
hold: {} violate {}.
hold:      { process(processor("Amy"),data(subject("Bob"),"X-Rays")),
            getConsent(processor("Amy"), subject("Bob"),

```

```

        process(processor("Amy"),data(subject("Bob"),"X-Rays")), }
    violate: { getConsent(processor("Amy"), subject("Bob"),
        process(processor("Amy"),data(subject("Bob"),"X-Rays"))) }.
    hold: {} violate {}

```

The meaning of each eFLINT specification is the linear trace through a semantic state-transition system induced by the input sequence of steps, where each step changes a persistent store of type-definitions. Example 4.15 showcases the incremental development of a collection of type-definitions in seven steps delimited by (.). The output reports the states resulting from each successive step:

1. Fact types `agent` and `data` are defined, each as a type-product of a specified sequence of other instance types (where `string` and `int` are inbuilt strings and integers). The `data` type is defined with a *derivation rule* clause (**Derived from ...**), which specifies conditions under which `data`-type instances are derived to *hold*, as an instance-expression over the current state. In this case, the derivation rule is simple: it enumerates two constant instances which hold.
2. The definition of `data` in the first step is extended with a new derivation rule. Consequently, a new, third `data`-type instance holds.
3. The `data` type definition is overwritten. All its prior instances are lost.
4. Two new `agent`-type identifiers are introduced. Then three fact- and one duty-type are introduced. The `getConsent` duty-type uses the prior `agent`-type identifiers to declare that the `getConsent` is a product-type with two distinctly identified `agent`-type fields. Note how the fields of duties are annotated distinctly from those of facts: specialised keywords mark the fields encoding the holder and claimant. This makes explicitly how each duty models a two-party, institutional, obligation relationship. Despite its complexity, this step preserves the prior, empty state, because the new derivation rule is not yet applicable.
5. A new `process` instance holds. Consequently, the derivation rule defined for `getConsent` applies for the first time, and so a new duty holds. Its violation condition is immediately met because its `process`-field does not hold.
6. Finally, a *condition clause* is added to the definition of `processor`. Consequently, no derivation rules apply, and there are no instances whatsoever.

**Simplifications** Our simplified eFLINT is a sub-language of the real eFLINT, so it is executable with the real eFLINT interpreter. We make some simplifying omissions.

- We omit instance- and predicate-expression operators including **Exists** and **Or**.
- We omit eFLINT's *postulation* statements, which add or remove individual instances in all subsequent states, overwriting previous, conflicting postulations.

- We omit *events*, *actions*, *invariants*, and *Booleans* which complement *duties* and *facts* in affording the definition of data types which are subtly specialised in syntax to mirror more legal concepts, and semantically in their interactions with violations and postulations. For our purposes, these omitted constructs can be sufficiently approximated by combinations of facts and duties.

#### 4.4.3.2 Simple Composition Control in eFLINT

Definition 4.24 embeds eFLINT in a simple language schema, such that it can be used for cooperative specification. This embedding prescribing a simple, dynamic validity criterion: the specification reports no violations.

**Definition 4.24** (violation-sensitive eFLINT). Let  $\mathcal{S}_e \triangleq \langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$ , where:

- $\mathcal{P} \triangleq$  *program* (Definition 4.22), the syntactic eFLINT specifications.
- $\mathcal{M} \triangleq$  *report* (Definition 4.23), reports of states' holding and violating instances.
- $\llbracket \cdot \rrbracket$  captures the dynamic semantics of eFLINT, unfolding a linear state trace, where each state separately enumerates the holding and violating instances.
- $\circ$  concatenates programs in  $\mathcal{P}$ .
- $\checkmark(p)$  recognises when 1.  $p$  satisfies static semantics of eFLINT: identifiers are bound by quantifiers or resolve to either identifier- or predicate-definitions, and expressions are well-typed, and 2. there is no violation in  $\llbracket p \rrbracket$ .

$\mathcal{S}_e$  affords cooperative specification through the manipulation of violated duties via statements (re)defining and extending fact- and duty-types. This definition of invalidity conflates normative violations (at the domain level) with unsuccessful cooperation (at the meta-level). On the one hand, this affords agents elegantly internalising cross-cutting concerns. This manifests as a wide variety of control triples  $\langle p_1, p_2, p_3 \rangle$ , including cases in which  $p_2$  contains no statements directly altering duties. Thus,  $\mathcal{S}_e$  lets agents cooperate to incrementally develop and update a model of a norm-compliant cyber-social system.

On the other hand, there are two problems. Firstly, it limits the applicability of eFLINT as a cooperative specification language. Precisely, agents cannot successfully model systems with normative violations at the domain level; this is problematic for modelling cyber-social systems in practice, where violations cannot be reliably avoided, so they are corrected and worked around instead. In these cases, it is necessary to distinguish failures of agents at the meta-level (cooperatively specifying the system) from normative violations of agents at the domain-level (inside the system being specified), even if these agents overlap. Secondly, even in the appropriate context, where agents aim to specify only violation-free systems, the characteristics

of the  $\mathcal{S}_e$  semantics creates a power-imbalance between contributors that impedes their productivity. Intuitively,  $\mathcal{S}_e$  thoroughly empowers later contributors, which earlier contributors experience as undermining their composition control, ultimately trivialising their influence on the final specification. Consider how the contributor of  $p_1$  has no control over each  $\llbracket p_1 \circ p_2 \rrbracket$ . Any derivation rule of type  $t$  in  $p_1$  can be effectively removed by  $p_2$  adding a (trivial) derivation condition to  $t$ , e.g., with  $p_2 \triangleq$  `Extend Duty getConsent Conditioned by False`. Moreover, any (clauses in) the definition of any type  $t$  in  $p_1$  can be effectively removed by  $p_2$  overwriting the definition of  $t$ , for example, with  $p_2 \triangleq$  `Duty getConsent Holder processor Claimant subject Related to process`.

Despite the problems these features of eFLINT pose, they cannot be simply removed, because they support the intended incremental refinements of the specification, for example, to let several stakeholders incrementally develop the model of the system. The real problem is that  $\mathcal{S}_e$  insufficiently captures which contributions the contributors permit. Example 4.16 demonstrates a more subtle case of this problem.

**Example 4.16** (later eFLINT contributors undermine prior contributors). Agents Amy, Bob, and Dan cooperate to develop a shared specification. First, Amy makes the following contribution, formalising the processing of data as a process-data relation, where processors are agents, and data is partitioned over subjects. Each subject can have multiple data-instances by identifying them with distinct strings.

```
Fact agent Identified by string
Fact data Identified by subject * string
Fact process Identified by processor * data
Fact started Identified by process
Placeholder subject For agent
Placeholder processor For agent.
```

Next, Bob formalises the legal notion of consent (to process data) as an agent-process relation. Bob also introduces the duty of processors to acquire consent for processing of a subject's data. Each duty instance is formalised as a processor-subject-process triple, specified to be held by the processor, and claimed by the subject. However, Bob does not (yet) specify any cases in which these duties are violated:

```
Placeholder consent For agent
Fact consent Identified by consent * process
Duty getConsent Holder processor Claimant subject Related to process
Derived from (Foreach process:
  getConsent(process.processor, process.data.subject, process))
Conditioned by Not(Holds(consent(subject, process))).
```

Next, Bob contributes an extension that defines when the duty to get consent is violated: when the process is started. Bob also amends the definition of `process` to ensure that Bob's prior `getConsent` derivation rule does not 'overlook' started processes.

In other words, Bob specifies that members of the `started` relation are not only of the `process` data type, but are also members of the `process` relation:

```
Extend Duty getConsent Violated when started(process)
Extend Fact process Derived from (Foreach started: started.process).
```

Dan makes the following, final contribution. The first statement is simple: Dan starts to process Amy's X-Ray Data. The other two statements remove Dan's duty to get consent for any data in general, and then (for good measure) grant Dan consent to process any data in general.

```
Extend Fact started
  Derived from started(process(processor("Dan"), data(subject("Amy"), "X-Rays")))
Extend Duty getConsent Conditioned by processor != agent("Dan")
Extend Fact consent Derived from (Foreach process:
  consent(process.data.subject, process) Where process.processor == agent("Dan")).
```

By preserving validity, the agents have cooperated successfully. But Dan has violated Bob's unspecified meta-level requirements (without overwriting any types).

1. Each agent is in control of their own consent.
2. Each existing duty to get consent is removed only by getting consent.

Bob experienced a lack of control over what Dan contributes. Dan experienced a lack of insight into what Bob permits or desires. For example, Bob and Dan each extend a type defined by another agent (Amy and Bob, respectively) with a derivation rule, failing to capture that Bob considers only the latter extension permissible.

Example 4.16 demonstrates the straightforward usage of eFLINT for cooperative specification.  $\mathcal{S}_e$  preserves eFLINT's extreme extensibility: the specification always remain flexible to new contributions. However, this comes at the cost of agents' power to control their peers. The final contributor has total control of the model.

#### 4.4.3.3 Improved Composition Control in eFLINT

We define  $\mathcal{S}'_e$  as a variant of  $\mathcal{S}_e$  which addresses some of its flaws.

**Definition 4.25** (extended eFLINT syntax). Let  $program'$  give the syntactic category of extended eFLINT programs (extending Definition 4.22), where:

$$program' := step'^* \quad step' := agent \text{ states } stmt'^+ .$$

$$agent := string \quad stmt' := stmt \text{ (Erroneous | Seal Derivations | Seal Conditions) }^*$$

**Definition 4.26** (extended eFLINT models). Let  $report'$  define the syntactic category of extended eFLINT reports (extending Definition 4.23), where:

$$report' := state'^* \quad state' := \{ hold: instSet , violate: instSet , error: instSet \}$$

**Definition 4.27** (apparent step contributor). Agent  $a$  *apparently contributes to*  $t$  in  $p : \text{program}'$  iff  $\exists(a \text{ states Extend Fact } t \dots) \in p \vee \exists(a \text{ states Extend Duty } t \dots) \in p$ .

**Definition 4.28** (improved eFLINT). Let  $\mathcal{S}'_e \triangleq \langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle \triangleq \mathcal{S}_e$ , but where:

- $\mathcal{P} \triangleq \text{program}'$ , the extended eFLINT specifications.
- $\mathcal{M} \triangleq \text{report}'$ , the extended eFLINT reports.
- $\llbracket p \rrbracket$  (as before) captures the dynamic semantics of eFLINT in the holding  $i_h$  and violating  $i_v$  instances of each state (**holds**:  $i_h$  **violate**:  $i_v$  **error**:  $i_e$ ). Let  $i_e$  capture the *errors*, the subset of  $i_h$  whose type definitions are marked **Erroneous**.
- Let  $\checkmark(p)$  iff 1. (as before)  $p$  satisfies the static semantics of eFLINT: identifiers are bound by quantifiers or resolve to either identifier- or predicate-definitions, and expressions are well-typed, 2. no statement is suppressed in  $p$ , 3. no type  $t$  has a **Seal Derivations** or **Seal Conditions** clause and respectively a **Derived from** or **Conditioned by** clause in a later step, 4. no **Derived from**  $e$  clause in  $p$  for a type  $t$ , with a **contributor** field of type **agent**, contributed by agent  $a$ , constructs  $s : \text{struct}$  which does not match  $t(a, \dots)$ , and 5. there is no error in  $\llbracket p \rrbracket$ .

Definition 4.25 defines *program'* as a syntactic extension of  $\mathcal{S}_e$ , *i.e.*, each program of the latter is a program of the former also. Definition 4.26 also gives these programs the same models, with respect to their holding and violating instances. However, Definition 4.28 gives a different meaning to the same violations. In  $\mathcal{S}'_e$ , violations are interpreted *only* at the domain-level, and their prior meta-level interpretation (invalidating the specification) is conferred to the new erroneous instances instead. This addresses the first flaw of  $\mathcal{S}_e$ :  $\mathcal{S}'_e$  affords the successful cooperative specification of cyber-social systems which exhibit normative violations (at the domain-level).

The second flaw of  $\mathcal{S}_e$  is addressed in  $\mathcal{S}'_e$  by adding a mix of the composition control mechanisms defined in the previous sections. These additions correspond to the new criteria (Points 2–5) for validity ( $\checkmark$ ) in Definition 4.28.

- Point 2 prohibits contributors from overwriting definitions, as this clearly lets later contributors trivialise prior contributions.
- Point 3 adapts the notion of static sealing from Definition 4.15 for eFLINT. In this case, two kinds of seal are distinguished, letting agents separately constrain **Derived from** and **Conditioned by** clauses. Intuitively, agents can separately control when instances can (not) be added or removed.
- Point 4 adapts Definition 4.11 to eFLINT, ensuring that instances with the **agent** field always reflect the contributor of the rule that derived them.
- Finally, Point 5 adapts the dynamic, error-sensitivity validity criterion for Datalog in  $\mathcal{S}'_D$  in Definition 4.10 to the eFLINT language.

All together, these new criteria for validity in  $\mathcal{S}'_e$  adapt a mix of static and dynamic composition control features originally used separately. Example 4.17 demonstrates how these come together to address the problems encountered in Example 4.16: Bob is able to express and enforce the intended meta-level requirements.

**Example 4.17** (successful cooperative eFLINT specification). We reconsider the cooperation between Amy, Bob, and Dan in Example 4.16. Amy begins, making the same contribution as before, just prefixed with `amy states`.

```
amy states
  Fact agent   Identified by string
  Fact data    Identified by subject * string
  Fact process Identified by processor * data
  Fact started Identified by process
  Placeholder subject For agent
  Placeholder processor For agent .
```

The second contribution, by Bob, is also essentially similar to how it was before. As in Amy's case, it is simply prefixed to mark its contributor. However, this time, Bob utilises the additional composition control mechanism to express the meta-level requirements that could not be expressed in Example 4.16. The agent parameterising the `consent` type is renamed to `contributor`, which has special significance: Bob can rely on the semantics of  $\mathcal{S}'_e$  to enforce the alignment between the contributor-argument of each `consent`-type instance with the identity of the agent that contributed the derivation rule that derived it. Essentially, Bob has specified that agents cannot create consent on each others' behalf. Finally, Bob has appended the new `Seal Conditions` clause to the definition of `getConsent`. This prevents subsequent contributions from suppressing the application of the existing clauses deriving `getConsent` instances.

```
bob states
  Placeholder contributor For agent
  Fact consent Identified by contributor * process
  Duty getConsent Holder processor Claimant subject Related to process
    Conditioned by Not(Holds(consent(subject, process)))
  Seal Conditions
  Derived from (Foreach process:
    getConsent(process.processor, process.data.subject, process)).
```

Everyone would agree that Dan's contribution from Example 4.16 is prohibited in this context for two reasons. Firstly, Dan cannot derive the consent of subjects of processed data. Secondly, Dan cannot remove `getConsent` duties by suppressing their derivation condition. Consequently, Dan chooses to make a different, permitted contribution. Dan experiences this as insight into what Bob considers permissible. Bob experiences this as control over what Dan contributes.

## 4.5 Designing Bespoke Cooperative Specification Languages

This section revolves around Table 4.1. Drawing from our experiences in Sections 4.5.2 and 4.5.3, we enumerate our *desirable qualities* for cooperative specification languages, filling in the first columns in the table. We explain our judgements of which of these qualities are present in Datalog, Alloy, and eFLINT, as they each come to their own resolutions of the tensions between these qualities, filling in the middle columns of the table. Finally, we fill in the last two columns by defining and characterising SEASO and Slick, in Sections 4.5.2 and 4.5.3 respectively, as new languages which are designed specifically to strike different compromises between the same qualities.

Recall that SEASO is precisely defined and demonstrated in Chapter 3, but here we re-frame it through our present notation and definitions, and in comparison to the other languages. In a nutshell, SEASO has Datalog-like syntax, and takes some semantic features of Alloy and eFLINT, but it is designed with static restrictions that guarantee the ‘robustness’ of models under composition. Slick is defined as a variant of SEASO that instead focuses on expressive and flexible dynamics. Slick is applied in Chapter 6, to express complex (meta-)policies regulating distributed agents’ access to sensitive data.

desirable quality	Datalog	Alloy	eFLINT	SEASO	Slick
$Q_1$ unique Boolean valuations	✓	✗	✓	✗	✗
$Q_2$ non-monotonic valuations	✗	✗	✓	✓	✓
$Q_3$ no semantic collapse	✓	✗	✓	✓	✗
$Q_4$ explainable truth	✓	✗	✗	✓	✓
$Q_5$ integer arithmetic	✗	✓	✓	✗	✗
$Q_6$ model computation terminates	✓	✓	✗	✓	✓
$Q_7$ heterogenous collections	✗	✗	✗	✗	✓

Table 4.1: An overview of the qualities  $Q_1$  to  $Q_7$  of cooperative specification languages, and which of these are present in Datalog, Alloy, eFLINT, SEASO, and Slick.

### 4.5.1 Desirable Qualities in Cooperative Specification Languages

**Unique Boolean Valuations ( $Q_1$ )** All of the languages we have studied define their semantics such that specifications can model logical formulae and assertions. Throughout this chapter, we have demonstrated how agents rely on this to model their requirements at every level. The simplest kind of semantic valuation maps each logical variable to exactly one *Boolean* value in  $\mathbb{B} \triangleq \{true, false\}$ . Consequently, each model fixes a Boolean answer to every query about the domain of discourse.

In Datalog and eFLINT, the model itself is such a valuation. Consider how in Example 4.6, the very first Datalog example, the truth or falsity valuation of

`no-access(amy, data1)` encodes the presence or absence, respectively, of a prohibition on agent Amy accessing Data1. And consider how in Example 4.17, the very last eFLINT example, the duty-type `getConsent` formalised the relation, enumerating each case of a data-subject with a claim that a data-processors has the duty to acquire the consent of the subject for processing the data.

Alloy also ultimately expresses meaning via Boolean valuations. But there is a complication: each model defines a *set* of instances, and each instance provides its own valuation of logical variables. Indeed, this is more general. But in our context, the presence of multiple distinct answers to the same queries means that agreement on a specification does not create agreement in the individual valuations, *i.e.*, the answers to queries. In Section 4.4.2, where we focused primarily on composition control, we were not concerned with ensuring that agents agree on every detail of the domain of discourse. But in practice, we expect this also to be desirable. For example, in Chapter 6, agents use agreement on action justifications to agree on which action effects they are permitted to realise outside of the system.

**Non-monotonic Valuations ( $Q_2$ )** Recall that agents encode (requirements on) domain properties in the model  $\llbracket p \rrbracket$  of the shared program  $p$ . They express changes in these requirements or properties indirectly, by adjusting  $p$ , *i.e.*, replacing it with some  $p \circ p'$ . Depending on the language, different models are reachable via these adjustments. We recognise the benefit of languages that let contributions change models' valuations *non-monotonically*, *i.e.*, from false to true and from true to false.

Contributions to Datalog programs introduce new new ways to infer the truth of logical variables on the condition of other truths. Famously, the truths in Datalog programs grow monotonically with the addition of rules. For example, `access(amy, data1)` is true in the model of Example 4.5, and no subsequent contribution can falsify this truth. In fact, truths are never falsifiable in Datalog. This makes (adjustments of) Datalog programs easy to reason about, but it makes them inflexible to changes in agents' encoded (requirements on) the properties of the domain. For this reason, it is common to extend Datalog with negative rule-conditions which are evaluated under weak negation, comparable to those in Clingo and eFLINT.

Alloy programs and models are more complex. Contributions to shared Alloy specifications add new constraints that remove instances. Unlike Datalog, this can include the removal of (models with) true valuations. Indeed, Alloy treats truth and falsity with remarkable symmetry; *e.g.*, facts can assert the truth or falsity of a formula, and any assertion can be (repeatedly) negated by a preceding `not`. But contributions are monotonic, but differently to those of Datalog: the constraints on Alloy instances only grow monotonically, or in other words, whether instances satisfy a given constraint can only change from true to false. In practice, this means that agents

cannot remove existing constraints, for example, to relax a prior requirement. For example, every adjustment of an Alloy specification that is already overconstrained (*i.e.*, it has no instances) is necessarily also overconstrained.

eFLINT reflects its design for flexibility. While its logical foundations are Datalog-like, it differs in two ways that let contributions non-monotonically change a model's valuations. Firstly, its derivation rules admit negated conditions of the form **Not**  $e$ , which are satisfied whenever the truth of  $e$  does *not* hold. Example 4.15 demonstrates this construct in action, where it causes the truth of `getConsent` instances to be removed if new `consent` instances become true, *e.g.*, because they are derived by newly-contributed derivation rules. Secondly, eFLINT lets existing derivation rules of an existing type  $t$  be effectively removed by the subsequent (re)definition of  $t$ . Example 4.16 demonstrates how these make models of eFLINT specifications extremely flexible to change (at the cost of prior contributors' control of the model).

**No Semantic Collapse ( $Q_3$ )** What should be the semantic valuation of contradictory logical formula? Specification languages that let users express contradictory requirements must address this problem. By their simplicity, Datalog rules cannot express contradictory requirements. But agents can make logically inconsistent assertions in both Alloy and eFLINT, but they address the problem differently.

In Alloy, contradictory requirements place unsatisfiable constraints on the instances. Alloy's approach is straightforward: these overconstrained specifications have no satisfying instances; appropriately, their model is the empty set of instances. This is an elegant solution that makes perfect sense in Alloy's usual context. But in our cooperative specification context, the potential for contributions to overconstrain the specification is problematic, because the resulting model is uninformative; recall Example 4.13, where the cause for overconstraint was subtle, and the absence of semantic value in the overconstrained model made it difficult for Dan to understand and explain the specification's invalidity. We call this a case of *semantic collapse*; the model has become uninformative and useless for further cooperation.

eFLINT provides an unorthodox solution to the threat of semantic collapse: the specified requirements are systematically weakened until they are satisfiable. Section 2.9 focuses on this facet of the eFLINT semantics. In summary, incremental computation steps collect truths by applying derivation rules, where negative conditions are evaluated in the moment, regardless if the conditions become falsified later. For example, **Derived from** `int(1) Where Not(int(1))` is applicable while `Not(int(1))` holds, whereafter `int(1)` holds and `Not(int(1))` no longer holds. Intuitively, eFLINT discards negative conditions from specified clauses which overconstrain any logical variables.

**Explainable Truth ( $Q_4$ )** Each Datalog truth has a clear explanation: it is the root of a *proof tree*, whose branches are program rules: incremental reasoning steps.

This is very desirable in our context, because each truth can (in principle) be traced back to its proof, whose reasoning steps can be examined and scrutinised, and each applied rule can be traced back to a particular contribution by a particular agent. In Chapter 6, we rely on this intuition to align these reproducible logical proofs with audits. Unfortunately, weak negation makes it more difficult to explain why a logical variable  $x$  is false, because – by definition – falsity of  $x$  is witnessed by no program rule, but rather by eliminating *all* rules which might otherwise prove  $x$ ;<sup>6</sup> there is no explanation except that no conceivable  $x$ -deriving rule was applicable.<sup>7</sup> Because we see no solution, we limit ( $Q_4$ ) to explaining just the truths.

Alloy’s semantics is radically different, because the enumeration of semantic instances is not driven by the syntactic constraints that the agents express. Consequently, the only explanation available for truths is that ‘they violate no constraints’.

In most cases, each truth in an eFLINT model (or *holding instance* in eFLINT parlance) is the result of an applied derivation rule, or was created as the effect of some event. These are explainable for the same reason as those of Datalog. But unfortunately, because eFLINT’s derivation procedure can retroactively falsify derivation rules that have already been applied, models may contain truths that are explainable by no satisfied rule. From another perspective, agents cannot express Alloy-style constraints by controlling the application of eFLINT derivation rules.

**Integer Arithmetic ( $Q_5$ )** Alloy and eFLINT implement typical operations on integers, such as testing integer pairs for equality (=) and ordering (<), or computing new integers via addition (+), subtraction (–), and so on.

Alloy lets users represent and compute integers as usual. This does not pose any problems for Alloy, because integer-expressions are only ever evaluated in constraints, which *filter* existing instances. But in eFLINT, derivation rules can use operators like + and – to construct new integers dynamically, which introduce new logical variables. This is useful in some contexts, because small programs can ‘build up’ arbitrarily large models by deriving new integers from others. Example 4.18 demonstrates.

**Example 4.18** (eFLINT instances from dynamically computed integers). This specification uses eFLINT’s inbuilt integer subtraction operator (–) to specify that each elapsed time step implies that all prior (nonzero) time steps have also elapsed.

```
Fact elapsed Identified by Int
  Derived from (Foreach elapsed: elapsed(elapsed - 1) Where 0 <= elapsed)
```

<sup>6</sup>Note how this resembles our characterisation of ‘weak negation’ as a language feature ( $Q_2$ ), despite our claim that Datalog does not have  $Q_2$ . Precisely, *weak negation* as a language feature typically refers to letting syntactic constructs *inside the language* reflect on this notion of falsity.

<sup>7</sup>Because weak negation proves  $\neg x$  from a proof that *there exists no way* to prove  $x$ , the proof of  $\neg x$  cannot be explained without reasoning about all applicable rules and ruling them out. This is what we mean by ‘less explainable’.

**Model Computation Terminates ( $Q_6$ )** Agents reason about the shared specification via the model that is evaluated by the  $[\cdot]$  function. It is undesirable if agents encounter programs which they cannot evaluate in practice, because (computing) the application of  $[\cdot]$  never terminates. The end result is comparable to semantic collapse ( $\neg Q_3$ ); the specification is useless because it impedes further cooperation. But there is a difference: agents can agree *whether* a model has semantically collapsed, but agents cannot generally agree whether a program’s evaluation will terminate.

The computation of Datalog models is famously simple and robust. Evaluation can apply rules and collect truths (interleavedly) to a fixpoint, because each truth is necessarily a finite combination of the finite concrete literals in the program text.

Because the AlloyAnalyzer itself is responsible for enumerating and checking the satisfiability of instances, the language addresses the threat of non-termination from the outset. The set of instances under consideration is always finite, because there is always a finite (implicit or explicit) cardinality constraint on each atomic set. The other (field) relations are in the product of the atomic relations. Consequently, evaluation necessarily terminates after the finite instances are considered.

Because eFLINT’s integer operators let the application of rules create new instances, there exist eFLINT programs of finite size which have infinite models. In practice, these programs cannot be evaluated. Example 4.19 demonstrates such a case. But we have demonstrated in Section 4.4.3 that agents do not need eFLINT’s integer arithmetic to model complex systems or complex requirements. If agents take care, the use of integer operators can preserve termination; for example, Example 4.18 uses integer arithmetic to compute new instances, and yet it terminates. But in principle, during cooperative specification, the threat of non-termination always looms.

**Example 4.19** (a finite eFLINT program with an infinite model). The following has an infinite model: in the only state,  $x(0)$  holds, so  $x(1)$  holds, so  $x(2)$  holds, *et cetera*.

```
Fact x Identified by Int Derived from x(0) Derived from (Foreach x: x(x + 1)).
```

**Heterogenous Collections ( $Q_7$ )** Each language uses has a notion of semantic *collection* whose elements are *quantified* by variables. Datalog, Alloy, and eFLINT have in common that the terms matching any given variable have the same internal structure. This structure is the most restricted in Datalog: variables quantify only over constants, which have no internal structure. Alloy variables are typed, and Alloy types are either atomics, in which case the instances are flat (as in Datalog), or they are records of atomics, in which case each element binds the same fields of the same types. eFLINT variables range over arbitrarily complex terms, because instances are *trees* of other instances, but the elements are still homogenous.

This homogeneity has indirect advantages. Firstly, it imposes a sort of simplicity on the agents, normalising the representation of any given concept. Secondly, the

regularity of terms can be exploited for the sake of efficiency. For example, because syntactic Datalog atoms have fixed types, Datalog interpreters can collect all the atoms of the same type into a homogenous vector, and iterate only over these when grounding (each variable inside) each atom of the same type.

But in our context, where contributions may represent significant commitments, the inflexibility of (variables iterating over) homogenous collections can impede cooperation. Example 4.20 demonstrates this with an eFLINT specification.

**Example 4.20** (the inflexibility of homogenous collections in eFLINT). Agent Amy formalises the notion of consent to the access of data as a relation, in terms of access, whose members are agent-data pairs. But Bob wants to distinguish access events on the same data by the same data at different *times*. But it is too late; eFLINT forced Amy to fix the internal structure of `access` before `consent` could be defined. Bob can overwrite the definition of the consent relation, but this would be undesirably destructive, discarding the type’s existing clauses and instances.

```
Fact agent Identified by String      Fact data Identified by String
Fact access Identified by agent * data  Fact consent Identified by agent * access
```

## 4.5.2 Seaso: Static and Robust Cooperative Specification

Here, we re-frame the SEASO language, which was designed and applied in Chapter 3 to the cooperative specification of data exchange systems. But now we formalise its relation to these concepts as they are defined, more precisely, in Section 4.2: language schemas, cooperative specification, and control triples.

**Definition 4.29** (SEASO syntax). Let *program* define the (syntactic) language of SEASO specifications, the input language of the SEASO interpreter tool.

$$\begin{aligned}
 \text{program} &:= \text{statement}^* & \text{mark} &:= (\text{seal} \mid \text{emit} \mid \text{decl}) \text{ domain} \\
 \text{statement} &:= \text{mark} \mid \text{defn} \mid \text{rule} & \text{defn} &:= \text{defn domain} ( ( \text{domain}^* ) )? \\
 \text{domain} &:= \text{lowercase character}^* & \text{rule} &:= \text{rule atom}^* ( ; - \text{antecedent}^* )? \\
 \text{variable} &:= \text{uppercase character}^* & \text{antecedent} &:= !? \text{atom} (= \text{atom}) \\
 \text{atom} &:= \text{variable} \mid \text{string} \mid \text{integer} \mid \text{domain} ( ( \text{atom}^* ) )?
 \end{aligned}$$

**Definition 4.30** (SEASO models). Let *denotation* define the (semantic) language of SEASO specifications, the output language of the SEASO interpreter.

$$\begin{aligned}
 \text{grAtom} &:= \text{string} \mid \text{integer} \mid \text{domain} ( ( \text{gratom}^* ) )? \\
 \text{denotation} &:= \{ \text{trues: grAtom}^* , \text{unknowns: grAtom}^* , \text{emissions: grAtom}^* \}
 \end{aligned}$$

Definitions 4.29 and 4.30 suffice to let SEASO model systems via denotations. But to use SEASO for cooperative specification, as it is defined in this chapter, we

must situate the language in a language schema, *i.e.*, we must make clear how the agents characterise the success of their cooperation. For example, we must decide how program *emissions* relate to its validity. Definition 4.31 defines  $\mathcal{S}_X$ , which lets emissions invalidate their programs, *i.e.*, agents agree that emissions are prohibited.<sup>8</sup>

**Definition 4.31** (SEASO with static sealing). Let  $\mathcal{S}_X \triangleq \langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$ , where:

- $\mathcal{P} \triangleq$  *program* (Definition 4.29), the syntactic SEASO specification language.
- $\mathcal{M} \triangleq$  *model* (Definition 4.30), the language of SEASO denotations.
- $\llbracket \cdot \rrbracket$  captures the dynamic semantics of SEASO, detailed in Section 3.3.4; it applies the program’s rules up to its alternating fixpoint, *i.e.*, implementing the well-founded semantics of van Gelder et al. [VGRS91].
- $\circ$  concatenates programs in  $\mathcal{P}$ .
- $\checkmark(p)$  recognises when 1.  $p$  satisfies the static semantics of SEASO (Section 3.3.3):  $p$  is *well-formed* (Definition 3.4), implying that terms are well-typed and no seal is broken, and 2.  $\llbracket p \rrbracket$  has no *emissions*, *i.e.*, no truths have types marked `emit`.

The mix of static and dynamic composition control mechanisms in  $\mathcal{S}_X$  is most similar to  $\mathcal{S}'_e$  from Section 4.4.3.3 (eFLINT with static seals and dynamic errors). Because  $\mathcal{S}_X$  recreates our application of SEASO for cooperative specification in Chapter 3, Section 3.4 already provides extensive examples of reasoning about and applying  $\mathcal{S}_X$ . Here, we characterise this language in comparison to the others.

**Qualities of Seaso** Table 4.1 shows how SEASO has a novel combination of the qualities of Datalog, Alloy, and eFLINT. Note that SEASO shares the largest number of features with Datalog. The idea behind SEASO was to sacrifice the qualities of Boolean valuations ( $Q_1$ ) and integer arithmetic ( $Q_5$ ) such that it can combine *all* of the other qualities of the existing languages at the same time. Firstly, this required identifying the fundamental tension between qualities  $\{Q_1, Q_3, Q_4\}$ . Consider the impossibility in even a simple case, where the semantics must give explainable Boolean valuations to  $x$  in a program asserting  $x \leftarrow \neg x$ . Any model must give a meaningful ( $Q_3$ ) and unique Boolean ( $Q_1$ ) valuation to  $x$ , but neither value is satisfactory; if  $x$  is true, this cannot be explained ( $Q_4$ ), and if  $x$  is false, the specified assertion was ignored.

SEASO adopts the *well-founded semantics* for logic programming [VGRS91], which sidesteps this fundamental problem by weakening  $Q_1$ : atoms are given a *ternary* valuation in  $\{true, false, unknown\}$ . The idea is that, in programs whose rules encode logical inconsistencies, each atom that is overconstrained by the ideal requirements

<sup>8</sup>The language schemas in this chapter clarify how to formalise the experimental SEASO semantic variants that are discussed informally in Section 3.5.7. For example, we introduce `flaw` as a syntactic like `emit`, add truths of `flaw`-marked typed (‘flaws’) to the semantic language alongside emissions, and tweak the definition of ( $\checkmark$ ) in  $\mathcal{S}_X$  such that flaws not emissions invalidate programs.

is given the *unknown* valuation, which is otherwise meaningless.<sup>9</sup> Desirably, this semantics otherwise aligns with the semantics of Datalog, where there are no logical inconsistencies, and so there are no unknown values, and atoms are true if and only if they are the head of a satisfied rule, and otherwise, they are false.

When variables with unknown values are ignored, the well-founded semantics aligns with the stable model semantics underlying Clingo and its weak negation. Consequently, as with eFLINT, contributors to SEASO programs can non-monotonically change the valuation of logical variables ( $Q_2$ ). Example 4.21 demonstrates.

**Example 4.21** (simple non-monotonicity in SEASO). SEASO program `rule 1 :- !2` makes `1` and `2` false and true, respectively. But adding `rule 2` flips both values.

Section 3.3.6 proves that SEASO’s semantics maps a unique finite model to each finite program in finite evaluation steps ( $Q_6$ ). This proof relies on the absence of integer arithmetic ( $\neg Q_6$ ) and heterogenous collections ( $\neg Q_7$ ). Intuitively, this proof relies on SEASO sharing the property with Datalog that there are finite truths to infer, because each is a finite combination of the finite constants in the program text.

All together, these qualities make SEASO ‘robust’ under composition; each program has a finite model. Section 3.4 already exhaustively demonstrates SEASO in application to cooperative specification. Example 4.22 demonstrates how this manifests in practice. The example also shows how the design of SEASO reflects its intended application in more subtle ways. For example, although the language still does not support heterogenous collections ( $Q_7$ ), types can be left unspecified such that, under composition, its variables have heterogenous *possible* structures.

**Example 4.22** (robustness of fragmented specifications in SEASO). The following SEASO program expresses the requirement that, under composition with any other programs, no data should be accessed without the consent of the subject. For flexibility under composition, the internal structure of agents, data, and context (of any access event) is left unspecified, so that it can be specified elsewhere. Access events are parametrised by context specifically to leave room for the introduction of additional arguments. For example, this avoids the problem shown in Example 4.20.

```
decl agent. data. context.
defn access(agent, data, agent, context). consent(agent, access). bad(access).
emit bad. rule bad(access(Accessor, Data, Subject, Context)) :-
    !consent(Subject, access(Accessor, Data, Subject, Context)),
    access(Accessor, Data, Subject, Context).
```

<sup>9</sup>From a different perspective, the well-founded semantics weakens  $Q_3$  by introducing semantic collapse at the granularity of individual variables, so that the model at large stays meaningful.

### 4.5.3 Slick: Dynamic and Flexible Cooperative Specification

Slick was developed as a variant of SEASO with the goal of emphasising flexibility in cooperative specification by introducing heterogenous collections ( $Q_7$ ) and laying the groundwork for re-introducing integer arithmetic ( $Q_5$ ). The cost is that Slick re-introduces the threat of semantic collapse ( $-Q_3$ ). To follow, we present the Slick syntax and semantics in comparison to SEASO, and then we discuss these changes to its qualities, and how they make Slick suitable for application in Chapter 6.

The Slick language and our interpreter implementation are (re)presented in Sections 6.5.2 and 6.5.4, respectively, before they are applied in Section 6.6.

**Definition 4.32** (Slick syntax). Let *program* give the Slick programs, where

$$\begin{aligned}
 \text{program} &:= \text{rule}^* & \text{rule} &:= \text{head} \ (\text{if } \text{body})? \\
 \text{head} &:= \text{atom} & \text{body} &:= (\text{not? } (\text{atom} \mid \text{cmp}))^{\text{and}*} \\
 \text{var} &:= \text{uppercase letter}^* & \text{cmp} &:= (\text{same} \mid \text{diff}) \{ \text{atom}^{\text{space}+} \} \\
 \text{const} &:= \text{lowercase letter}^* & \text{atom} &:= \text{const} \mid \text{var} \mid ( \text{atom} ) \mid \text{atom} \ \text{space} \ \text{atom}^{\text{space}+}
 \end{aligned}$$

**Definition 4.33** (Slick models).  $\text{model} := \{ \text{true}: \text{atom}^*, \text{unknown}: \text{atom}^* \}$ .

Because they are no longer needed to ensure termination, Slick relaxes SEASO's requirement that terms must be typed. In fact, the Slick definition is dramatically simplified by doing away with types altogether. In fact, all arguments to each Slick atom are symmetrical; there is no need for a constant in the initial position to fix the atom's type. Rule conditions bind *any* atoms with the matching structure ( $Q_7$ ). For example,  $x$  matches both  $a$  and pair  $a \ b$ , but  $x \ b$  matches only the latter.

Note that the concrete syntax of Slick exploits the interchangeability of atom arguments to make atoms more readable: without types, brackets are only used for parenthesis, *i.e.*, to group atoms together into (sub)atoms. The symbolic tokens ( $:-$ ),  $(,)$ , and  $!$  are replaced with textual **if**, **and**, and **not**. Consequently, Slick rules become very pleasant to read (in our opinion) by approximating natural language.

The lack of types, and the emphasis on flexibility, it made sense to give Slick a dynamic definition of invalidity. Precisely, defines  $\mathcal{S}_Y$  with the same definition of validity as was originally shown with Datalog in Definition 4.10. In practice, the same mechanism is significantly more flexible, because of Slick's weak negation and heterogenous collections. Section 6.6 demonstrates Slick in action extensively. Here, it suffices to give a small example to contrast Slick with the other languages.

**Definition 4.34** (Slick with dynamic invalidity). Let  $\mathcal{S}_Y \triangleq \langle \mathcal{P}, \mathcal{M}, \llbracket \cdot \rrbracket, \circ, \checkmark \rangle$ , where:

- $\mathcal{P} \triangleq \text{program}$  (Definition 4.32), the syntactic SEASO specification language.
- $\mathcal{M} \triangleq \text{model}$  (Definition 4.33), the language of SEASO denotations.

- $\llbracket \cdot \rrbracket$  captures the dynamic semantics of Slick, which are the same as SEASO, but where terms have no preceding type, and where antecedents quantify any truth with the matching structure (binding each variable to the according subterm).
- $\circ$  concatenates programs in  $\mathcal{P}$ .
- $\checkmark(p)$  holds iff `error` is not true in  $\llbracket p \rrbracket$ .

**Example 4.23** (heterogenous quantification in Slick). This approximates Example 4.22 in Slick: Bob observes the following contribution by Amy. But Amy now uses the structure of atoms and dynamic invalidity to express their (meta)requirements.

```
error if not Subject consents to (Accessor accesses Data of Subject when Context)
and Accessor accesses Data of Subject when Context.
```

**Forcing Termination via Semantic Collapse ( $Q_6$ )** As in eFLINT, the evaluation of Slick programs threatens non-termination, because rules can construct more atoms than those that occur in the finite program text. In Slick, even without integer arithmetic, this arises from the lack of types, which are the basis for the proof of termination in SEASO. But we can prove non-termination in Slick with one witness: the infinite set of truths  $(f (f (f ( \dots x \dots )))$  are each true in  $\llbracket (f x) \text{ if } x. x. \rrbracket$ . No implementation can terminate if it is tasked with exhaustively enumerating these truths. In more subtle programs, the threats of termination are more difficult to detect, and just checking for validity can approximate the halting problem.

To ensure termination, the inference algorithm underlying the Slick semantics differs from that of Datalog (and most related languages) in one detail. Inference halts after some static bound on intermediate computation is exceeded. This is comparable to how Alloy bounds the search for instances by bounding the cardinality of atomic sets. Precisely, inference halts after an intermediate inference step applies a rule that constructs a truth whose depth exceeds a statically configured *maximum depth* (e.g., 10). This adapts the approach presented in [Cho95], but there is a crucial detail: exceeding the bound also invalidates the program!

Intuitively, this mechanism prefers semantic collapse ( $\neg Q_3$ ) over non-termination ( $\neg Q_6$ ). Recall the benefit of  $Q_6$  over  $Q_3$ : although agents still encounter meaningless specifications, they always agree *which* specifications are meaningful. This suffices for the intended application of Slick in Chapter 6, where agents explore many specifications and do not commit to any one in particular, so it is acceptable if some meaningless specifications are encountered and then discarded.

Finally, we note that this approach to ‘forcing’ termination is widely applicable; it is easy to imagine how it can be adapted to many logic-based languages such as eFLINT. But we note that the adaptation must be careful to preserve the functional abstraction of the semantics ( $\llbracket \cdot \rrbracket : \mathcal{P} \rightarrow \mathcal{M}$ ), because this is the basis for agents’

agreement on the domain of discourse following from their agreement on the program. As an example to the contrary, consider forcing termination of inference in Slick or eFLINT by imposing a statically agreed timeout duration in trying to compute a model; assuming the usual unreliability of computation time on realistic hardware, agents can disagree on the meaning of the shared program  $p$  because they disagree on whether attempts to compute  $\llbracket p \rrbracket$  time out (under different runtime conditions)!

## 4.6 Related Work

This section overviews related work and remarks on opportunities for using related works to inform the cooperative specification of data exchange systems in the future.

### 4.6.1 Composition Control in General-Purpose Languages

Many long-established general purpose programming languages have language features that are essential for protecting fundamental abstractions. One example is the *visibility-* or *access-modifiers* in languages including Rust and Java. These have no impact on the dynamic semantics, but only introduce static errors if identifiers are accessed out of the specified scope. Most obviously, visibility modifiers protect abstractions, thus hardening programs against the accidental introduction of bugs.

In general, various lines of research offer various abstractions and controls for the change and maintenance of programs. We consider research on *module systems*, *version control*, and *API design*, broadly construed. For example, Haskell and Rust both organise program constructs into modules which are primarily structured hierarchically, like file systems. But there are differences in detail that dramatically change how programs can be changed. For example, modules in Rust can be secondarily linked together via module-aliases that traverse up and down the hierarchy, which relaxes the structure of large projects, but can make inner modules brittle to refactorings of outer modules. As another example, both the Rust and Haskell languages themselves offer limited expressivity to document requirements on program constructs, *e.g.*, function signatures. But because of its pure functional foundations, these requirements are often formulated in the Haskell community very precisely in terms of equality constraints; we have in mind the precise formulation of the *monad laws* in the Haskell standard library.<sup>10</sup> In comparison, Coq has the same functional foundations with a very similar encoding of monads in its standard library, but it is natural to express the monad laws within Coq itself.<sup>11</sup> The usual trade-off between expressive power and reasoning cost arises here also. For example, research

<sup>10</sup>See <https://hackage.haskell.org/package/base-4.21.0.0/docs/Control-Monad.html>.

<sup>11</sup>See <https://rocq-prover.org/doc/v8.12/api/coq/Monad/module-type-Def/index.html>.

explores auxiliary tools for systematically enforcing the laws of Haskell type classes via property-based testing [JJA12] or automatic theorem proving [AJT19]. All of these approaches and languages offer ideas for compositional control mechanisms.

The strength of many of these approaches is their static nature, which makes them (relatively) easy to reason about, and explainable in terms of syntactic constructs under programmer control. Visibility modifiers directly inspired the *seals* presented in Section 4.3.2.4. They have also inspired other works for similar reasons. For example, [Ala11] adapts them for use in securing object-oriented databases.

Many existing languages and features remain to be investigated for applicability to cooperative specification. For example, we hypothesise that the notion of *mode* in the Mercury language [SHC94] captures a simple, useful, static abstraction. In Mercury, modes prescribe the input/output modalities of arguments in logical predicates, enabling their compilation to efficient, imperative procedures [DHR01].

## 4.6.2 Smart Contract Languages

Smart contract languages are oriented around the multi-party definition and use of *smart contracts*, which encode inter-agent powers as cryptographically secured, replicated, executable programs [Sza97, GIM<sup>+</sup>18]. We are particularly inspired by two characteristics of these languages.

Firstly, smart contract languages are designed around the understanding that shared contracts represent *commitments*; they are meaningful and useful because they are not trivially retracted. For example, contracts justify costly and sensitive work. A major consequence is that, in this context, it makes less sense to rely on destructive refactorings to amend specifications. Instead, there is greater emphasis on extensible specifications and anticipating and avoiding conflicts and contention. This view is evident in our notion of cooperative specification, which frames change as an inherently constructive process via specification composition.

Secondly, smart contract languages emphasise the capturing of inter-agent power dynamics. This is achieved by internalising agents in the specifications themselves, such that they may be systematically reasoned about in relation to other domain-level concepts. For example, the *Daml* smart contract language (presented in an archived article [BFH<sup>+</sup>23]) reflects *signatories* of smart contracts as the *agent* argument of the corresponding contract construct. To some extent, this pattern is even observable in languages targeting a broader notion of contract. For example, eFLINT provides *actor* as an inbuilt type which parametrises each user-defined *action*-type by default [vBLvDvE20], affording their relation to domain-level concepts via eFLINT types and instances, as usual.

### 4.6.3 Powerful Formalisms and Formal Verification

Many communities develop (understandings of) complex software systems by encoding them in powerfully expressive formalisms, to capture complex requirements and to automate complex reasoning processes. For example, by encoding system behaviour as dependently-typed definitions, the Coq theorem prover can verify complex properties.

Compared to works in these communities, the specification languages we have considered are conservative in the power of their semantics and the extent of their automation. To some extent, this is incidental, and can be improved; future work can improve the expressivity of Alloy and eFLINT so they can capture new requirements. However, it is also advantageous to minimise the expressive power of our specification languages up to the requirements of the use case. This choice minimises conceptual burden on human users developing specifications, and minimises the cost of automated reasoning activities. Moreover, it leaves room for more powerful languages and systems to assist in the development and reasoning about simple specifications. For example, we see potential in using the (more powerful) Clingo language to partially automate the search for desirable contributions to a cooperative specification expressed in the (less powerful) Datalog language. Section 2.7.2 demonstrates how Clingo can be used to search for combinations of atoms to select and make become true. We see this as a small step away from tasking Clingo to search for combinations of (condition-free) Datalog rules to be contributed to the shared specification. However, we have yet to explore the details of this approach.

### 4.6.4 Abstract Argumentation Frameworks

Abstract argumentation frameworks in the style of Dung [Dun95] capture logical systems in which different logical conclusions can conflict, precisely, as an argument-argument *attack* relation. The ASPIC+ framework<sup>12</sup> builds new relations (like argument-preorder  $\leq$ ) and properties (like *contrary*) for characterising logical systems [vdTV14]. Its use is in enabling reasoning about key *rationality postulates* of particular systems [PM12], which can be understood as characterising a notion of usefulness. Works like [MP11] are related, investigating argument-preferences as a means to resolve conflicts.

These works complement our own; we share the premise of composing declarative specifications whose meaning is robust to the presence of conflicts and changes. Appropriately, our motivations also often overlap. For example, [ML08] investigates abstract argumentation in the context of conflicts between agent-local desires, and system-wide norms. Consider how Dan's desire to acquire consent conflicted with the

---

<sup>12</sup>It appears that ASPIC and ASPIC+ originally did not abbreviate anything.

legal norms formalised by Bob in Example 4.17. We see promise in more thoroughly studying and incorporating the methods and tools of abstract argumentation in the development and application of cooperative specification languages.

## 4.7 Discussion

**The Framework** In this chapter, we have demonstrated the potential and challenges in (modifying languages to) the application of specification languages to the *cooperative specification* problem. Our definitions in Section 4.2 gave the cooperative specification problem (*e.g.*, which was first encountered in Chapter 3) a more formal framing. The subsequent chapters demonstrated the benefits of our general approach of turning specification languages into meta-specification languages. The benefits followed from this approach systematically connecting the specification of the usual domain-level with meta-level concepts, concerning the agents' changes of the shared specification.

Firstly, this approach reinforced the idea that agents' meta-level requirements can be specified with the same rigour, and in the same way, as their domain-level requirements. This discourages ad-hoc solutions, *e.g.*, under-specifying the power of particular agents to change facets of the specification. With our approach, it was always clear that the meaning of meta-level requirements were ultimately precisely defined via their framing by our notion of language schema. If agents agreed on the shared specification language, they necessarily agreed on which contributions to the shared specification were permitted in any context.

Secondly, our approach clarified how to formalise requirements that related the domain- and meta-levels, which we expect to be necessary in practice. While our dynamic composition control mechanisms took this to the extreme, it was present even in the simpler examples using more conservative, static composition control mechanisms. Consider Example 4.14, where the legal expert Bob connected the domain-level concept of *consent* with the meta-level notion that only Bob is permitted to define the role of *consent* in the specification.

**The Space of Available Languages** Sections 4.3 and 4.4 focused on how the features of the language interact to determine its *composition control*, determining which meta-level requirements the specifications can capture.

We have explored the effects of making minor alterations to the syntax and semantics of these languages, and how these can dramatically improve the suitability of the language to cooperative specification, without significantly changing the way it models the usual domain-level concepts. For example, recall how eFLINT without and with language extensions (in Sections 4.4.3.2 and 4.4.3.3, respectively) let agents control the rules their peers contributed via static 'seals'.

We observed that even a relatively simple Datalog language afforded various composition control features, affording different use cases. For example, Datalog with the static ‘sealing’ mechanism (in Section 4.3.2.4) captured meta-level requirements in terms that agents could easily understand, and which were easily computed, while Datalog with dynamic invalidity (in Section 4.3.2.2) captured complex meta-level requirements, but were more complex for agents to reason about, both because the meta- and domain-level concepts were inter-connected.

We observed that the same composition control mechanisms were recognisable even when applied in specification languages which differed significantly in their syntax and semantics. For example, Datalog, Alloy, eFLINT, and Seaso, each afforded the expression of essentially similar static ‘seals’ to control composition, with comparable results. But we also observed that some combinations were more natural and useful than others. For example, it was not clear how to adapt the dynamic invalidity mechanism (used in Datalog, eFLINT, SEASO, and Slick) to Alloy, which did not share the logic-programming foundations of the other languages.

**Qualities of Cooperative Specification Languages** Section 4.5 reflected on our exploration of language features. Section 4.5.1 distilled our findings to a set of seven *desirable characteristics* of cooperative specification languages. Each is discussed, motivated, and exemplified via our findings from adapting and applying Datalog, Alloy, and eFLINT to the cooperative specification problem.

Most immediately, these qualities gave us a means to compare and evaluate our findings across the different languages. We reflected on the relationships between these qualities: which combinations are possible and desirable. This guided our design of our own, bespoke languages, SEASO and Slick, for cooperative specification, which are applied in Chapters 3 and 6, respectively. We hope that others in the future will be inspired by these findings to further develop useful languages and tools for cooperative specification, perhaps specialised for different cases, proposing different composition control features, or inspired by different specification language features.

## 4.8 Ongoing & Future Work

**Specialised eFLINT Variants** In Section 4.5, we evaluated the qualities of eFLINT for cooperative specification. Previously, we discussed which of our desirable qualities eFLINT has, but it also has some other qualities that we did not identify explicitly, because they are more difficult to generalise to other languages. For example, eFLINT’s tree-like representation of logical variables, *e.g.*, which are also present in Clingo, were found to be convenient in compositionally building up models, so this feature carried over into Slick and SEASO. eFLINT has other qualities that we expect to be practical

in practice. Notably, its foundations in Hohfeld’s framework for legal reasoning has afforded its more extensive application to the formalisation of legal norms.

We are interested in mitigating eFLINT’s flaws by exploring alterations to its semantics. We see two promising directions. Firstly, we want to evaluate our the new semantics for eFLINT that are the focus of Chapter 2. Intuitively, this moves eFLINT closer to Alloy, by collapsing models in the presence of logical inconsistencies ( $Q_3$ ), rather than introducing truths that cannot be explained ( $Q_4$ ). Of course, this introduces a flaw of Alloy to eFLINT, but we expect it to arise in fewer realistic cases, as a consequence of eFLINT’s Datalog-like logic programming rules, *i.e.*, in contrast to Alloy’s constraint-based constructs. Secondly, we want to experiment with adapting the well-founded semantics to eFLINT. Intuitively, this moves eFLINT closer to SEASO, by ‘containing’ logical inconsistencies as unknown valuations. Recall that this effectively sacrifices Boolean valuations ( $Q_1$ ) for explainable truth ( $Q_4$ ).

**Specialised Datalog Variants** Like eFLINT, Datalog has a constructive approach to prescribing relations, which lays the groundwork for capturing meta-level requirements whose violations are easily diagnosed and understood. Datalog has been thoroughly researched by many people over many years. For example, some of this research is summarised in [MTKW18, KK22]. Consequently, there are many formalised Datalog variants to choose from.

Section 4.5 already explored our development of Datalog-like languages: SEASO and Slick. These adopt an essentially Datalog-like syntax and inference semantics, which was found to have considerable applications, despite the simplicity of its language definition and the brevity of its programs. This made Datalog-like languages easy to experiment with and reason about. But much work remains to evaluate SEASO and Slick in practice, and to explore other Datalog-like languages.

**Enforcement** Our notion of cooperative specification resembles dynamic enforcement (*e.g.*, via access control) when cooperative specification is interleaved with the execution of the specified system. Our work was done with this possibility in mind. Chapter 6 explores one approach by effectively adapting our present *language schemas* into the *JustAct* framework, which includes the syntax, models, and validity of a policy language as before, but also relates the dynamics of the agents, *e.g.*, which policy messages they have created and communicated.

## 4.9 Conclusion

In this chapter, we have presented the potential and the challenges of agents cooperating to formalise their requirements of a complex, distributed system. We consider

data exchange systems in particular, whose nature exacerbates the usual difficulties in expressing and composing requirements.

We give a technical framing of our approach to this problem as *cooperative specification*, where agents incrementally compose their contributions to a shared program, careful to preserve its *validity*. We showed that existing languages Datalog, Alloy, and eFLINT take us part of the way to satisfactory solutions. But we observe their limitations in capturing important ‘meta-level’ requirements, which concern the ways agents may change the specification itself. For example, legal experts permit data subjects to express their own consent to the processing of their data, but prohibit data consumers from arbitrarily claiming the consent on the data subjects’ behalf, or removing the need for consent. The benefit of this approach is that it re-frames the unfamiliar problem of controlling changes to the specification as the familiar problem of defining the syntax and semantics of specification languages.

We explored various ways to define validity, evaluating their utility via our metric of *composition control*: situations where agents can intercept particular changes to the specification. We demonstrate minor changes to the languages. For example, we added a new clause to the eFLINT language, which afforded new use cases by improving composition control, without significantly changing the expression of the usual domain-level concepts. We compared the different choices of languages that we explored by identifying seven *desirable qualities* ( $Q_1$ – $Q_7$ ) of cooperative specification languages. We recognised how each language made a unique trade-off between these qualities. We designed and demonstrated the SEASO and Slick languages specifically to strike novel trade-offs. We briefly demonstrated how this enabled novel use cases, summarising their extensive usage in Chapters 3 and 6, respectively.

We identified several opportunities for future work to continue developing specialised cooperative specification languages. We hope to leverage related works to ease the development of languages and specifications, to the end of improving the productivity of the specification process, and the fruitfulness of its results. This contributes to the vision of inter-organisational data exchange systems executed by autonomous, cooperating agents that enforce their shared requirements, even when requirements are changed continuously, throughout their enforcement.

**General Takeaways** This chapter gave a formal framing of the cooperative specification problem that is explored throughout this thesis. We take away that the nature of the cooperation depends on the features of the specification language, and that this requires balancing trade-offs between various desirable qualities. But in general, some composition control is essential for enabling practical use cases.



# Formal Foundations for Reowolf: Multi-Party Sessions via Synchronous Protocol Programming

## Abstract

The Reowolf project developed *connectors* as a replacement of two-party network sockets for multi-party communication in next-generation internet applications. Users control connectors via protocols in the bespoke *protocol description language* (PDL), which is based on synchronous languages such as Reo and Esterel. The novelty lies in the emphasis on dynamism: users refine protocols throughout their execution.

We formalise the semantics of PDL, distinguishing dual notions of protocol behaviour: *accepted* behaviour is highly (de)compositional and specifies *what* communication is allowed, while *constructed* behaviour arises from protocol execution and accounts for *how* execution steps interdepend and interleave via messages sent and received. Toward machine-checking the correctness of the Connector Runtime reference implementation, we specify the API and correctness criteria of PDL Runtime systems.

**Basis of this Chapter** This chapter is primarily based on the COORDINATION2025 conference article [ELHA25] and adopts its supplementary artefact [ELH25]: the Coq specification of the PDL semantics, PDL Runtime systems, and proofs of their properties. Sections 5.1 and 5.2 adapt the motivations of the Reowolf project and the API of the resulting connectors from a position paper presented at the FACS2019 conference [EH19]. Figure 5.1 is adapted from the project documentation [EH24].

## 5.1 Introduction

Owing to decades of fruitful research, synchronous coordination languages like Reo [Arb04, Arb16] and Esterel [BG92, FYTF19] enable the specification of complex, multi-party communication behaviour using synchronous protocols. These protocols

specify essential ordering and data-dependencies of messages and computations, but stop short of fixing the low-level implementation details. Consequently, protocols are abstract and compositional, affording powerful systematic analysis and verification against high-level properties such as fairness and deadlock freedom. Such protocols are often applied in the coordination of communications between software components. Extensive literature explores this usage, *e.g.*, compiling protocols expressed in Reo [DA18, JA15, JSS<sup>+</sup>12, JSS<sup>+</sup>14] and Esterel [Edw02a, Edw02b, EZ07, PBEB07] into low-level ‘glue code’ which mediates communications between software components. The resulting applications exhibit complex coordinated behaviour, but they can be systematically reasoned about via their protocols, *e.g.*, to verify properties. Also, this approach decouples components, making them easier to maintain and reuse.

The *Reowolf project* investigated the application of Reo-like synchronous protocols to the coordination of communications in distributed systems in general and on the Internet in particular [EH19]. In this new context, where distributed peers come and go, it is impractical to collect and compile all peers’ requirements as a single *session protocol* in overview before the session begins. Instead, the idea is that communication behaviour unfolds, one synchronous round at a time, in between changes to the protocol. Peers can come and go, and adjust the session

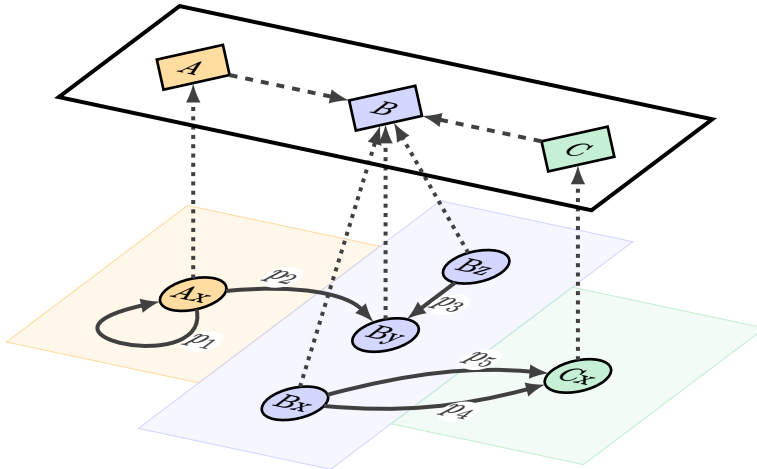


Figure 5.1: An example Reowolf connector runtime session, visualised graphically. Distributed connectors (diamonds) each connects one user to the session. Users have partial views of the ‘data plane’ (below), where the session protocol consists of primitive components (circles), whose management is partitioned over the connectors (dotted arrows). These primitives specify nondeterministic execution, message passing at shared ports (solid arrows), interleaved with local computations. The runtime system coordinates and synchronises via control messages in the ‘control plane’ (above). This figure is adapted from Figure 6.1 of the Reowolf documentation [EH24].

protocol to reflect their changing requirements. This idea is implemented in the *Connector Runtime* middleware system, whose API is the *connector*. Like BSD-style network sockets, each connector maintains the user-facing abstraction of *session*, by transparently automating the underlying control communications and resource management. Unlike sockets, connectors are extensively programmable. Users communicate entirely via Reowolf’s *Protocol Description Language* (PDL), which is based on Reo and other high-level synchronous languages, but was co-designed with connectors. The Reowolf project contributed the design, Rust implementation, and benchmarking of the Connector Runtime, as detailed in extensive technical documentation [EH24]. To frame our present contributions, we overview the usage of connectors in Section 5.2: we lay out the connector API and user requirements as Invariants 5.1 to 5.4. Together, these characterise the separation of concerns between the users and the distributed Connector Runtime system. Figure 5.1 visualises the result: the users rely on the distributed Connector Runtime to continuously unfold consistent synchronous communication behaviour, from their ad-hoc and piece-meal specification of behaviour as a growing network of stateful, non-deterministic, (a)synchronous message-passing programs.

Presently, we formalise the main contributions of the Reowolf project, and clarify their fundamental properties. Firstly, we give a **definition of PDL** in Section 5.3. Fundamentally, we distinguish dual notions of (communication) behaviour: *accepted* behaviours specify the set of all possible behaviours, while *constructed* behaviours restrict the accepted behaviours, by considering the execution of the session at large, forcing a causal ordering of messages, and interleaving the sending of messages with local computations. We prove key properties of PDL as Theorems 5.1 to 5.4. For example, Theorem 5.4 formalises an essential idea: behaviours constructed from any composite protocol are accepted by each part. Secondly, we give a **specification of PDL Runtimes**, in Section 5.4, as runtime systems which let users interleave protocol execution and refinement. We define their correctness as preserving Properties 5.1 to 5.4, which connect Invariants 5.1 to 5.4 to the PDL semantics. We implement an illustrative PDL Runtime, show that it satisfies all properties but *completeness* (Property 5.4), and discuss approaches to implementing PDL Runtimes that preserve these properties. Section 5.5 enumerates lines of future work: defining and evaluating various PDL protocols and runtimes. This includes the next steps toward verifying the correctness of the Connector Runtime implementation. Finally, we compare Reowolf’s PDL and connectors to related works (Section 5.6), before we conclude with a summary (Section 5.7).

All definitions and proofs in this chapter are formalised and machine-checked with the Coq proof assistant [BC13]. The resulting artefact is available at <https://>

[//zenodo.org/records/14936561](https://zenodo.org/records/14936561). See Appendix E for a breakdown of the artefact. This includes an explanation of its parameters (*i.e.*, assumptions) and a table detailing the correspondences between terms in the chapter vs. the artefact. Some readers may be interested in Appendix F, which provides a gentle introduction to the Coq language in general and our formalisation in particular.

## 5.2 User Communication Sessions via Connectors and PDL

Users of Reowolf connectors, physically distributed over the Internet, have a consistent experience of a shared communication session. To each user (and their application), the session is represented by a local, socket-like *connector* object. As with sockets, users interact only indirectly, via direct actions on their connector. The API provides users with three fundamental operations on their connector:

1. A user can **join a (possibly new) session** by participating in a set of rendezvous at chosen IP addresses. For example, Amy and Bob rendezvous at IPv4 address 65.49.82.45 to create and join a two-party session. Each rendezvous creates a new, persistent *port*, a logical place of message exchange.
2. A user can **reflect on the current behaviour** of the session, which fixes the messages exchanged so far. Thus, users can read received messages.
3. A user can **refine the session protocol**  $p$  by replacing it with its composition with user-provided protocol  $p'$ . We say the user *refines*  $p$  or *injects*  $p'$  into the session. Section 5.3 elaborates on PDL, but in short, protocols specify behaviour: protocols are stateful, message-passing programs, whose nondeterministic execution constructs behaviour. Consequently, injecting protocols is comparable to spawning new worker actors, processes, or threads.

Users are ultimately concerned with protocols and sessions via their *behaviour*. We adopt the notion and nomenclature of (finite) behaviours from Reo, *e.g.*, in [Arb11]: each behaviour specifies the message, if any, observed per logical place (port) per discrete chronological instant (round), for  $n : \mathbb{N} \triangleq \{0, 1, 2, 3, \dots\}$  rounds completed so far, *i.e.*, rounds are what we call the *indices* of behaviours. While the Connector Runtime defines message data as byte sequences for parity with UDP datagrams and IP packets, instead, we model message data  $\mathcal{D}$  with the natural numbers ( $\mathbb{N}$ ). *Ports*  $\mathcal{P}$  are an arbitrary data type that users understand as shared variables.

**Notation 5.1** (basic notations for data and messages).

- $\uplus$  denotes the disjoint set union, *i.e.*,  $A \uplus B \triangleq A \cup B$  is defined iff  $A \cap B = \emptyset$ .
- $\star$  marks the absence of any message at a port;  $\star$  is not a message.
- $\mathbb{N} \triangleq \{0, 1, 2, 3, \dots\}$  denotes the natural numbers.

**Definition 5.1** (data, messages, and behaviours).

$$\begin{array}{ll}
 p, p', p_1, \dots : \mathcal{P} & \text{(ports)} \\
 d, d', d_1, \dots : \mathcal{D} \triangleq \mathbb{N} & \text{(data)} \\
 m, m', m_1, \dots : \mathcal{M} \triangleq \mathcal{P} \rightarrow \mathcal{D} \uplus \{\star\} & \text{(\underline{message map or messages})} \\
 M, M', M_1, \dots : \mathcal{M}^* \triangleq \text{list}(\mathcal{M}) & \text{(\underline{message map lists or behaviours})}
 \end{array}$$

The connector API also lets users follow the conventions of the socket API: messages can be sent and received on the fly. Precisely, these calls are transparently translated into *oneshot* protocols, which are injected as usual; **oneshot** is a protocol combinator, defined precisely later (in Figure 5.2), which lets users immediately exchange messages. For example, Amy sends data 4 to Bob via port  $p$ ; transparently, the connector injects **oneshot**(send {4}  $\leftrightarrow p$ ) into the session.

Users expect their runtime to preserve Invariants 5.1 to 5.4. Section 5.3 defines their underlying terms (*e.g.*, ‘*accepts*’) and Section 5.4 addresses their preservation.

**Invariant 5.1** (consistency). Users’ observations of the messages remain consistent with some behaviour  $[m_1, m_2, \dots, m_n]$ , which is only extended over time, *i.e.*, messages cannot change retroactively. To support PDL Runtimes that are physically distributed (discussed in Section 5.5.1), we consider consistency to be preserved even if 1. some messages are hidden from some users, and 2. some users have not yet observed the latest round’s messages  $m_n$ .

**Invariant 5.2** (acceptance). Each protocol presently injected by users *accepts* the future session behaviours w.r.t. the PDL semantics. This safety property is the basis of the users’ power to meaningfully participate in communications.

**Invariant 5.3** (provenance). Each sent message has a sensible *provenance*: it can be traced back to some (user that injected some) protocol that sent it at some instant. Users may be oblivious to the identities or locations of these senders (or their peers in general). Nevertheless, provenance is necessary in practice, because

- it enforces the role of the runtime to facilitate real user communications, for example, rather than fabricating arbitrary messages, and
- it ensures that users can be ultimately held accountable for the messages they send, *e.g.*, to enforce data protection regulations.

**Invariant 5.4** (productivity). Within the constraints of Invariants 5.1 to 5.3, some next communication round is always eventually completed. Intuitively, this requires that the PDL Runtime does not deadlock or overlook specified behaviour.

### 5.3 Protocol Description Language (PDL)

This section defines the PDL, affording the expression of composable protocols. Ultimately, the meaning of protocols is in their two key notions of behaviour: the behaviours each protocol *accepts* and the subset of those it *constructs*.

In practice, users rely on some syntactic protocol composition operator which is symmetric, associative, and commutative with respect to the PDL semantics. For simplicity, instead, we represent protocols as decomposed into sets of their constituent *primitive* protocols. Thus, protocol composition is set union ( $\cup$ ) and *empty protocol*  $\{\}$  is the identity element with respect to composition. By design, our formalism gives this protocol sensible semantics; later we show that it always has the same behaviour as the primitive protocol **loop sync**.

**Notation 5.2.**  $2^\phi$  to denotes the powerset of any set  $\phi$ .

**Definition 5.2** (primitive protocols, composite protocols, and their behaviours).

$$\begin{aligned} s, s', s_1, \dots & : \mathcal{S} && \text{(primitive protocol \underline{statements})} \\ S, S', S_1, \dots & : 2^{\mathcal{S}} && \text{(composite protocols, \textit{i.e.}, \underline{statement sets})} \\ \textit{constructs} \subseteq \textit{accepts} \subseteq 2^{\mathcal{S}} \times \mathcal{M}^* & && \text{(semantic protocol-behaviour relations)} \end{aligned}$$

#### 5.3.1 PDL Syntax and Small-Step Semantics

Each primitive PDL protocol is executed as a *worker*: a pair of a local *memory* in  $\Sigma$  (persistent, local stores mapping *variables* to data) and a PDL *statement* in  $\mathcal{S}$ . Like ports, variables ( $\mathcal{V}$ ) are arbitrary. Workers are essentially comparable to the usual notions of agent, thread, or process that mutates its local memory.

**Definition 5.3** (workers and their variable stores).

$$\begin{aligned} v, v', v_1, \dots & : \mathcal{V} && \text{(variables)} \\ \sigma, \sigma', \sigma_1, \sigma_2, \dots & : \Sigma \triangleq \mathcal{V} \rightarrow \mathcal{D} && \text{(variable \underline{stores} or memory)} \\ w, w', w_1, w_2, \dots & : \mathcal{W} \triangleq \Sigma \times \mathcal{S} && \text{(\underline{workers} executing primitive protocols)} \\ W, W', W_1, W_2, \dots & : 2^{\mathcal{W}} && \text{(sets of \underline{workers} executing composite protocols)} \end{aligned}$$

Figures 5.2 and 5.3 give the precise syntax and semantics of PDL primitives, respectively, which realise a granular notion of execution as a mostly-conventional while-language. Precisely, *update* gives a small-step operational semantics for PDL:

*update* is the transition function over workers, parametrised by given messages. Equivalently, fixing the messages yields a fragment of update of type  $\mathcal{W} \rightarrow \mathcal{W}$ : how each worker reacts to the messages. Moreover, each *update* is labelled by a *synchronicity* (Definition 5.4), which recognises whether it was *synchronous*.

**Definition 5.4** (synchronicity of a worker update step).

$$\delta, \delta', \delta_1, \delta, \dots : \Delta \triangleq \{\mathcal{X} \text{ (synchronous)}, \checkmark \text{ (asynchronous)}\}$$

The *asynchronous* ( $\mathcal{X}$ ) steps are largely conventional for while-languages. Example 5.1 demonstrates a typical imperative program: asynchronous steps implement Euclid’s algorithm for the greatest common divisor of integers in  $v$  and  $v'$ , using  $v''$  to swap  $v$  and  $v'$ , and terminating as **done** with the result in  $v'$ .

**Example 5.1** (asynchronous local computation: greatest common divisor of  $v$  and  $v'$ ).  
**while**  $0 < (v \% v')$  **do** (**write**  $(v \% v') \hookrightarrow v''$ ; **write**  $v' \hookrightarrow v$ ; **write**  $v'' \hookrightarrow v'$ ).

Statements **send** and **recv** are also asynchronous, letting workers remember and react to the messages in their environment. These statements’ roles in *update* are similar: they move data between memory and messages. Later, Section 5.3.4 distinguishes their roles as one might expect: **send** is the *only* way to put new messages at ports. In this role, **send**  $E$  expresses a *nondeterministic choice* whose outcomes are identified by the elements of  $E$ . Example 5.2 demonstrates conditional, synchronous, and non-deterministic message passing: forwarding any number except 21 from port  $p$  to port  $p'$ , but incremented by 1 or 2, chosen non-deterministically.

**Example 5.2** (message passing and non-determinism, within a synchronous round).  
**recv**  $p \hookrightarrow v$ ; **if'**  $(v \neq 21)$  **send**  $\{v + 1, v + 2\} \hookrightarrow p'$ .

The **sync** statement is uniquely *synchronous* ( $\checkmark$ ). Intuitively, it delimits successive synchronous rounds. This is insignificant from the perspective of (updating) a single worker, but in Section 5.3.2, we formalise the intention: **sync** also acts as a synchronous barrier: **all** workers synchronise together, ‘committing’ the round’s

$o : \mathcal{O} := + \mid - \mid * \mid \div \mid \% \mid \wedge \mid \vee \mid < \mid = \mid \neq$	(binary operators)
$e, e_1, e_2 : \mathcal{E} := d : \mathcal{D} \mid v : \mathcal{V} \mid e_1 \ o \ e_2 \mid \neg e$	( $\mathcal{D}$ -type expressions)
$s, s_1, s_2 : \mathcal{S} := \mathbf{done} \mid \mathbf{sync} \mid \mathbf{write} \ e \ \hookrightarrow \ v \mid s_1 ; s_2$	(computation statements)
$\mid \mathbf{if} \ e \ s_1 \ \mathbf{else} \ s_2 \mid \mathbf{while} \ e \ \mathbf{do} \ s$	(control flow statements)
$\mid \mathbf{send} \ (E : 2^{\mathcal{E}}) \ \hookrightarrow \ p \mid \mathbf{recv} \ p \ \hookrightarrow \ v$	(communication statements)
-----	
$\mathbf{if}' \ e \ s \triangleq \mathbf{if} \ e \ s \ \mathbf{else} \ \mathbf{done}$	$\mathbf{loop} \ s \triangleq \mathbf{while} \ 1 \ \mathbf{do} \ s$
$\mathbf{assert} \ e \triangleq \mathbf{if}' \ \neg e \ \mathbf{loop} \ \mathbf{done}$	$\mathbf{oneshot} \ s \triangleq s ; \mathbf{loop} \ \mathbf{sync}$

Figure 5.2: The syntax of primitive PDL protocol-expressions  $\mathcal{E}$  and -statements  $\mathcal{S}$ .

$$\begin{aligned}
\text{update}(m, \langle \sigma, \mathbf{done} \rangle) &\triangleq \langle \mathcal{X}, \langle \sigma, \mathbf{done} \rangle \rangle \\
\text{update}(m, \langle \sigma, \mathbf{sync} \rangle) &\triangleq \langle \surd, \langle \sigma, \mathbf{done} \rangle \rangle \\
\text{update}(m, \langle \sigma, \mathbf{write } e \hookrightarrow v \rangle) &\triangleq \langle \mathcal{X}, \langle \sigma[v := \text{eval}(\sigma, e)], \mathbf{done} \rangle \rangle \\
\text{update}(m, \langle \sigma, \mathbf{if } e \text{ } s \text{ } \mathbf{else } s' \rangle) &\triangleq \begin{cases} \langle \mathcal{X}, \langle \sigma, s \rangle \rangle & \text{if } \text{eval}(\sigma, e) \neq 0 \\ \langle \mathcal{X}, \langle \sigma, s' \rangle \rangle & \text{otherwise} \end{cases} \\
\text{update}(m, \langle \sigma, \mathbf{while } e \text{ } \mathbf{do } s \rangle) &\triangleq \begin{cases} \langle \mathcal{X}, \langle \sigma, \mathbf{done} \rangle \rangle & \text{if } \text{eval}(\sigma, e) = 0 \\ \langle \mathcal{X}, \langle \sigma, s ; \mathbf{while } e \text{ } \mathbf{do } s \rangle \rangle & \text{otherwise} \end{cases} \\
\text{update}(m, \langle \sigma, \mathbf{recv } p \hookrightarrow v \rangle) &\triangleq \begin{cases} \langle \mathcal{X}, \langle \sigma[v := m(p)], \mathbf{done} \rangle \rangle & \text{if } m(p) \neq \star \\ \langle \mathcal{X}, \langle \sigma, \mathbf{recv } p \hookrightarrow v \rangle \rangle & \text{otherwise} \end{cases} \\
\text{update}(m, \langle \sigma, \mathbf{send } E \hookrightarrow p \rangle) &\triangleq \begin{cases} \langle \mathcal{X}, \langle \sigma, \mathbf{done} \rangle \rangle & \text{if } \exists e \in E, m(p) = \text{eval}(\sigma, e) \\ \langle \mathcal{X}, \langle \sigma, \mathbf{send } E \hookrightarrow p \rangle \rangle & \text{otherwise} \end{cases} \\
\text{update}(m, \langle \sigma, s_1 ; s_2 \rangle) &\triangleq \begin{cases} \langle \delta, \langle \sigma', s_2 \rangle \rangle & \text{if } s_1 = \mathbf{done} \\ \langle \delta, \langle \sigma', s'_1 ; s_2 \rangle \rangle & \text{otherwise} \end{cases} \\
&\quad \text{where } \langle \delta, \langle \sigma', s'_1 \rangle \rangle \triangleq \text{update}(m, \langle \sigma, s_1 \rangle)
\end{aligned}$$

Figure 5.3: Primitive protocol update semantics, defined via  $\text{update} : \mathcal{M} \times \mathcal{W} \rightarrow \Delta \times \mathcal{W}$  in terms of the (omitted) expression evaluation function  $\text{eval} : \Sigma \times \mathcal{E} \rightarrow \mathcal{D}$ .

behaviour, and advancing to the next. Example 5.3 demonstrates; it accepts (and constructs only) two rounds of behaviour:  $[\{p \mapsto 1\}, \{p \mapsto 2\}]$ . Section 5.5.2 discusses a generalisation, where a *subset* of workers synchronise.

**Example 5.3** (completing two rounds).  $\mathbf{send } \{1\} \hookrightarrow p; \mathbf{sync}; \mathbf{send } \{2\} \hookrightarrow p; \mathbf{sync}$ .

### 5.3.2 Composite Protocols and (A)synchrony

Definition 5.5 generalises worker-update steps to sets of workers  $W, W', \dots : 2^{\mathcal{W}}$ .  $W \xrightarrow{m} W'$  asynchronously updates **one** worker in  $W$ , and  $W \xrightarrow{m} W'$  synchronously updates **each** worker in  $W$ . We omit  $W, W'$ , or  $m$  from terms of this form only when they are arbitrary or clear in context. For example,  $(W \rightarrow)$  denotes the assertion that some worker in  $W$  can perform some asynchronous update.

**Definition 5.5** (synchronous ( $\Rightarrow$ ) and asynchronous ( $\rightarrow$ ) worker-set steps).

$$\frac{\text{update}(m, w) = \langle \mathcal{X}, w' \rangle}{W \uplus \{w\} \xrightarrow{m} \{w'\} \cup W} \quad \frac{}{\emptyset \xrightarrow{m} \emptyset} \quad \frac{W \xrightarrow{m} W', \text{update}(m, w) = \langle \surd, w' \rangle}{W \uplus \{w\} \xrightarrow{m} \{w'\} \cup W'}$$

**Definition 5.6** (blockage). We call asynchronous updates that preserve the worker(s) *blocked*. I.e., where  $\text{update}(m, w) = \langle \mathcal{X}, w \rangle$  and  $W \xrightarrow{m} W$ .

Lemma 5.1 identifies a property of the worker-set update semantics. Because it is one of the few properties underlying our main theorems, it is one of the few characteristics of the primitive PDL semantics that we consider to be essential. Here, it emerges from our definition of  $\text{update}$  in Figure 5.3. Section 5.5.2 discusses variants of the primitive PDL semantics which preserve this property.

Intuitively, Lemma 5.1 asserts that asynchronous<sup>1</sup> updates do not let workers observe or remember the *absence* of messages at ports. The condition  $W \neq W'$  excepts the case where adding messages *unblocks* the workers, *i.e.*, the update step induces some change to the worker, *e.g.*, by changing its memory. By definition, the history of blocked update steps taken to reach any worker(s) had no effect on the worker(s). Example 5.4 demonstrates: adding message  $p \mapsto 1$  unblocks the worker. It still takes an asynchronous step, but the worker is changed: the new worker remembers the observed message, but it cannot remember the prior blockage.

**Definition 5.7** (message map subsumes  $\leq$ ).  $m \leq m' \triangleq \forall p : \mathcal{P}, m(p) \in \{m'(p), \star\}$ .

**Lemma 5.1** (asynchronous message-determinism). Any asynchronous step to distinct workers is preserved by adding messages (at empty ports).

For each  $W \neq W'$  and  $m \leq m'$ ,  $W \xrightarrow{m} W'$  implies  $W \xrightarrow{m'} W'$ .

*Proof.* By Definition 5.5,  $m$  asynchronously updates some  $w \in W$  to some  $w' \neq w$ , and it suffices to show property  $P$ : that  $m'$  does likewise. We distinguish the cases of the first statement  $s$  of  $w$ . If  $s = \mathbf{send} \ v \hookrightarrow p$  or  $s = \mathbf{recv} \ p \hookrightarrow v$ , then because  $w \neq w'$ , necessarily  $m(p) \neq \star$ . Because  $m' \geq m$ ,  $m'(p) = m(p)$ , so  $P$  holds. For any other  $s$ , the update of  $w$  is independent of the messages, so  $P$  holds.  $\square$

**Example 5.4** (unblocking a worker blocked at  $\mathbf{recv} \ p$  by adding  $1 : \mathcal{D}$  at port  $p$ ).

$$\frac{\mathit{update}(m[p := \star], \langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle) = \langle \mathcal{X}, \langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle \rangle}{\{\langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle\} \xrightarrow{m[p := \star]} \{\langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle\}} \quad (\text{blocked})$$

$$\frac{\mathit{update}(m[p := 1], \langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle) = \langle \mathcal{X}, \langle \sigma[v := 1], \mathbf{done} \rangle \rangle}{\{\langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle\} \xrightarrow{m[p := 1]} \{\langle \sigma[v := 1], \mathbf{done} \rangle\}} \quad (\text{unblocked})$$

### 5.3.3 Linear Execution Traces and Accepted Behaviour

Where Section 5.3.2 generalised the execution of one worker to several, we finally generalise one execution step to linear *traces* of contiguous execution steps. We denote traces by concatenating  $\rightarrow$  and  $\Rightarrow$  steps. Finally, let  $\rightarrow^*$  denote the transitive reflexive closure of  $\rightarrow$ . *E.g.*,  $W \xrightarrow{m^*} W'$  abbreviates  $W \xrightarrow{m} \xrightarrow{m} \dots \xrightarrow{m} W'$ , *i.e.*, a trace of asynchronous steps labelled with the same message map  $m$ .

Definitions 5.7 to 5.11 define key concepts: *traces* of contiguous steps are delimited into *rounds*, such that the  $n^{\text{th}}$  synchronous step ends the  $n^{\text{th}}$  round. We generally restrict our attention to *growing* rounds (Definition 5.11), where each message persists to the end of the round. Intuitively, each growing round incrementally collects messages in  $m$  until  $\xrightarrow{m}$  makes  $m$  observable as *behaviour* (Definition 5.10).

<sup>1</sup>In fact, our definition of *update* affords generalising Lemma 5.1 to all unblocked updates. This is included in our artefact. But Lemma 5.1 suffices for our main results.

**Definition 5.8** (trace). A trace is a sequence of contiguous  $\rightarrow$  or  $\Rightarrow$  steps.

**Definition 5.9** (round). A round is a trace with only one synchronous step, at the end. For example, round  $W \xrightarrow{m_1} \xrightarrow{m_2} \xrightarrow{m_3} \xrightarrow{m_4} W'$  consists of four steps.

**Definition 5.10** (behaviour). The behaviour of a trace is the concatenation of its synchronized messages. *E.g.*, the behaviour of  $\xrightarrow{m_1} \star \xrightarrow{m'_1} \xrightarrow{m_2} \star \xrightarrow{m'_2}$  is  $[m'_1, m'_2]$ .

**Definition 5.11** (growing round). A round is growing iff  $m \leq m'$  holds for each  $m$  and  $m'$  labelling consecutive steps in the round.

Definition 5.12 defines *acceptance*, the first of two notions of protocol behaviour. Acceptance is *imperative* in the sense that behaviour arises from traces of contiguous ('stateful') execution steps. However, acceptance is also *declarative* in the sense that messages are growing but otherwise unspecified. Acceptance treats protocols as behaviour-recognisers or -acceptors, *i.e.*, we check for acceptance given protocol-behaviour pairs. Acceptance is suitable for reasoning about execution in an environment open to messages sent externally, *e.g.*, to reason about *what* executions of a known protocol are possible in an unknown system.

**Definition 5.12** (accepted behaviour). Protocol  $S$  accepts the behaviour of each trace consisting of only growing rounds, starting from  $\{\langle \emptyset, s \rangle \mid s \in S\}$ .

**Example 5.5** (examples of protocols accepting behaviours).

- **{loop sync}** synchronises each round immediately, regardless of the messages, like the empty protocol  $\{\}$ , *i.e.*, leaving behaviour unconstrained.
- **{loop done}** and **{done}** perform an endlessly unproductive sequence of asynchronous updates, so they accept only the zero-round behaviour  $[\ ]$ .
- **{rcv  $p \hookrightarrow v$  ; sync}** and **{rcv  $p \hookrightarrow v$  ; send  $\{v\} \hookrightarrow p$  ; sync}** accept  $[\ ]$  and any  $[m]$  as long as  $m(p) \neq \star$ , *i.e.*, there must be *some* message at  $p$ .
- **{send  $\{0, 1\} \hookrightarrow p$  ; rcv  $p \hookrightarrow v$  ; assert  $v$  ; sync}** sends either 0 or 1 at port  $p$ , but then only synchronises if 1 was chosen. This demonstrates a useful pattern: workers can process and suppress behaviour before it is observable.

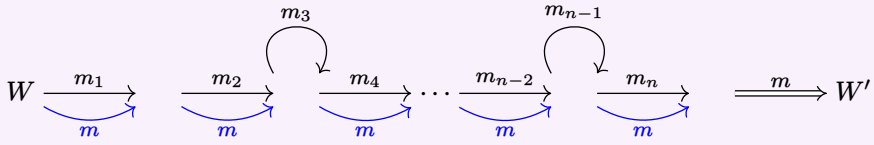
Recall Lemma 5.1: asynchronous steps are preserved by adding to the messages. Lemma 5.2 follows: for each growing round, there is a constant round with the same behaviour. Thus, all behaviours are sufficiently characterised by focusing only on *constant* rounds. These have useful properties, namely Lemma 5.3: workers in constant rounds cannot interact, because the messages are fixed, so each workers' updates proceed independently. Finally, Theorem 5.1 results: the change to each worker each round is entirely determined only by its behaviour. This has practical applications. For example, one can reliably 'replay' the execution of any set of

workers, arbitrarily many rounds into the future, given only the behaviour they produced. Section 5.5.1 discusses how this lets (distributed) PDL Runtimes explore traces concurrently, and use behaviours to identify (desirable) constant rounds.

**Definition 5.13** (constant round). A round is constant in messages  $m$  or  $m$ -constant iff each of its steps is labelled  $m$ . Note that constant rounds are growing.

**Lemma 5.2** (growing to constant). For each growing round synchronising  $m$ , a round exists with the same start and end workers, but which is *constant* in  $m$ .

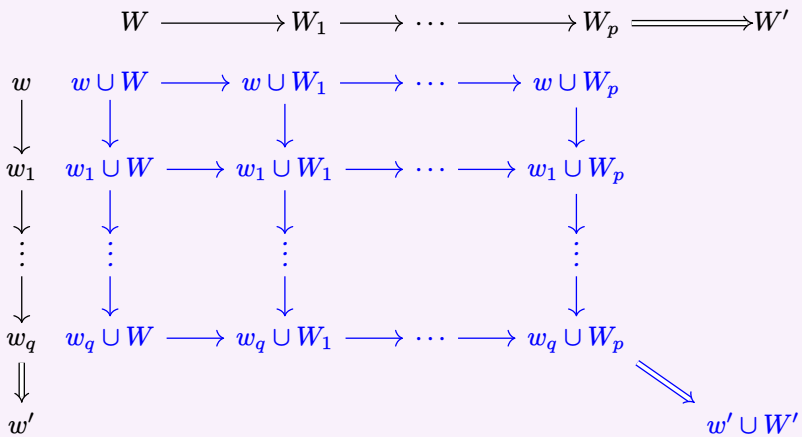
*Proof.* Each given step  $W_k \xrightarrow{m_k} W_{k+1}$  (in black) is ignored if  $W_k = W_{k+1}$ , i.e., it is a loop. Otherwise, as  $m_k \leq m$ , Lemma 5.1 lifts it to  $W_k \xrightarrow{m} W_{k+1}$  (in blue).



□

**Lemma 5.3** (constant confluence). For any workers  $W$  and messages  $m$ , all  $m$ -constant rounds starting from  $W$  necessarily end in the same workers.

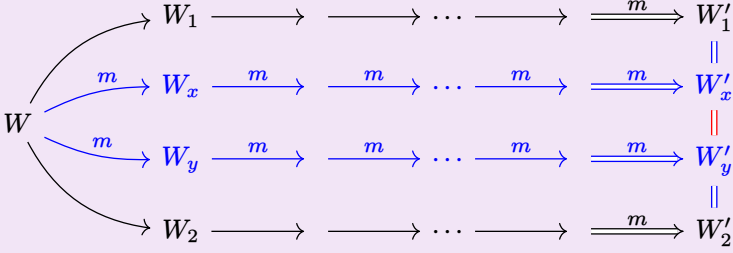
*Proof.* We prove the confluence of rounds from  $W$  by induction on the workers in  $W$  in an arbitrary order. The base case of  $W = \emptyset$  is trivial: the only step is  $\xrightarrow{m} \emptyset$ . In the inductive case, rounds  $W \xrightarrow{m^*} \xrightarrow{m}$  are confluent. We omit the proof that rounds  $\{w\} \xrightarrow{m^*} \xrightarrow{m}$  are confluent, where there is only one worker to *update* each step. Take arbitrary rounds from  $W$  to  $W'$  and from  $\{w\}$  to  $\{w'\}$  respectively (in black). Their steps can be arbitrarily interleaved (in blue), but must end in  $W' \cup \{w'\}$ . For legibility, the implicit  $m$  is omitted from each step in the figure below.



□

**Theorem 5.1** (round determinism). Growing rounds with the same start ( $W$ ) and behaviour ( $m$ ) necessarily have the same end ( $W'_1 = W'_2$ ).

*Proof.* By Lemma 5.2, for each given round (in black), some constant round has the same behaviour (in blue), whose ends are the same (in red) by Lemma 5.3.

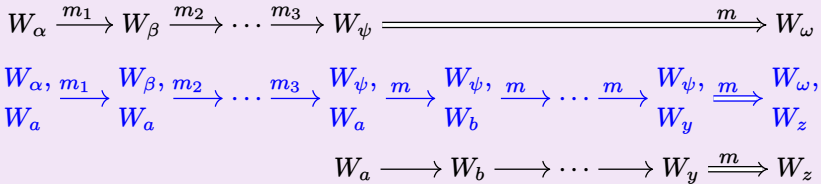


□

Finally, PDL satisfies strong (de)compositionality properties: by Theorems 5.2 and 5.3, the behaviours accepted by any protocol is the set-intersection of those of its constituent protocols. First of all, we expect PDL programmers to pervasively rely on Theorem 5.2 to control the behaviour of their session, because they can rely on the runtime system to avoid executions whose behaviour is not accepted by their injected protocols. The specification of accepted behaviours is similar to *constraint programming*: injecting protocols constrains the future behaviour. Users can also exploit this (de)compositionality during reasoning and verification, for example, to predict possible messages before the session begins.

**Theorem 5.2** (compositionality). Given growing rounds  $W_\alpha \xrightarrow{*m} W_\omega$  and  $W_a \xrightarrow{*m} W_z$ , there exists a growing round  $(W_\alpha \cup W_a) \xrightarrow{*m} (W_\omega \cup W_z)$ .

*Proof.* Given rounds  $r_1, r_2$  (in black), construct a new one (in blue) from the union of all initial workers. Repeat asynchronous steps of  $r_1$  (with the given messages), and then  $r_2$  (but replace messages with  $m$  using Lemma 5.1). Synchronise  $m$ .



□

**Theorem 5.3** (decompositionality). Given growing round  $W_\alpha \cup W_a \xrightarrow{*m} W$ , there exist worker sets  $W_\omega$  and  $W_z$  and growing rounds  $(W_\alpha \xrightarrow{*m} W_\omega)$  and  $(W_a \xrightarrow{*m} W_z)$ , such that  $W = (W_\omega \cup W_z)$ .

*Proof.* The proof is inductive, building two traces using the given round's steps, from the first to the last. We consider an arbitrary step from  $W_\psi \cup W_y$  with some  $m'$  to some  $W$ , where, by the inductive hypothesis,  $W_\alpha \xrightarrow{*} W_\psi$  and  $W_a \xrightarrow{*} W_y$  are constructed so far. If the step is asynchronous, exactly one worker  $w$  is updated to some  $w'$ , so  $W = ((W_\psi \cup W_y) \setminus w) \cup \{w'\}$ .  $W_\psi$  or  $W_y$  must contain  $w$ . Let it be  $W_\psi$ ; we omit the symmetric case. Construct  $W_\alpha \xrightarrow{*} W_\psi \xrightarrow{m'} (W_\psi \setminus \{w\}) \cup \{w'\}$  (in black) and leave the other round intact (in blue). Indeed  $(W_\psi \setminus \{w\}) \cup \{w'\} \cup W_y = W$ . Only the final step is synchronous; by definition of  $\Rightarrow$ , there exist  $W_\omega$  and  $W_z$  updated by  $W_\psi \xrightarrow{m'} W_\omega$  and  $W_y \xrightarrow{m'} W_z$ , where  $W = W_\omega \cup W_z$  (in red).

$$\begin{array}{ccccccc} W_\alpha, & \xrightarrow{m_1} & W_\beta, & \xlongequal{\quad} & W_\beta, & \xlongequal{\quad} & W_\beta, \dots & W_\psi, & \xrightarrow{m} & W_\omega, \\ W_a & \xlongequal{\quad} & W_a & \xrightarrow{m_2} & W_b & \xrightarrow{m_2} & W_c & \dots & W_y & \xrightarrow{m} & W_z \end{array}$$

□

### 5.3.4 Behaviour Constructed from PDL Protocols

Recall that acceptance uses the *update* function to compute workers from the prior workers, given any growing messages. Construction (Definition 5.15) also introduces *offered* (Definition 5.14), which computes messages from the prior workers.

Construction is suitable as a model for executing the system in its entirety, notably, by PDL Runtimes, in order to compute behaviour from the session protocol. Construction ensures the desired characteristics. Each observed message has a sensible provenance (Invariant 5.3). Moreover, each step is computable from the prior step, despite the domains of port, protocol, and data being arbitrarily large (thus, infeasible to enumerate). Figure 5.4 visualises the incremental construction of rounds, unfolding the (finite) options of each next messages and workers, by applying functions *offered* and *update*, in alternating fashion: the  $n^{\text{th}}$  messages are selected from the  $n^{\text{th}}$  workers' offers, the  $n + 1^{\text{st}}$  workers are selected from the  $n^{\text{th}}$  workers updated with the  $n^{\text{th}}$  messages, and so on.

**Definition 5.14** (offered messages).  $w$  offers to send  $d$  at  $p$  iff  $\langle p, d \rangle \in \text{offered}(w)$ .

$$\text{offered}(\langle \sigma, s \rangle : \mathcal{W}) : \text{list}(\mathcal{P} \times \mathcal{D}) \triangleq [\langle p, \text{eval}(\sigma, e) \rangle \mid (\text{send } E \hookrightarrow p) = s, \forall e \in E]$$

**Definition 5.15** (constructive round). A growing round is constructive iff, for each step from some  $W$  labelled  $m'$ , where  $m$  labels the prior step if it exists and  $m = \emptyset$  otherwise, either  $m = m'$  or there exist  $w \in W$  and  $\langle p, d \rangle \in \text{offered}(w)$ , such that  $m[p := d] = m'$ , in which case we say  $w$  sends  $d$  at  $p$ .

**Definition 5.16** (constructive behaviour). Protocol  $S$  constructs the behaviour of each trace from  $\{\langle \emptyset, s \rangle \mid s \in S\}$  consisting of only constructive rounds.

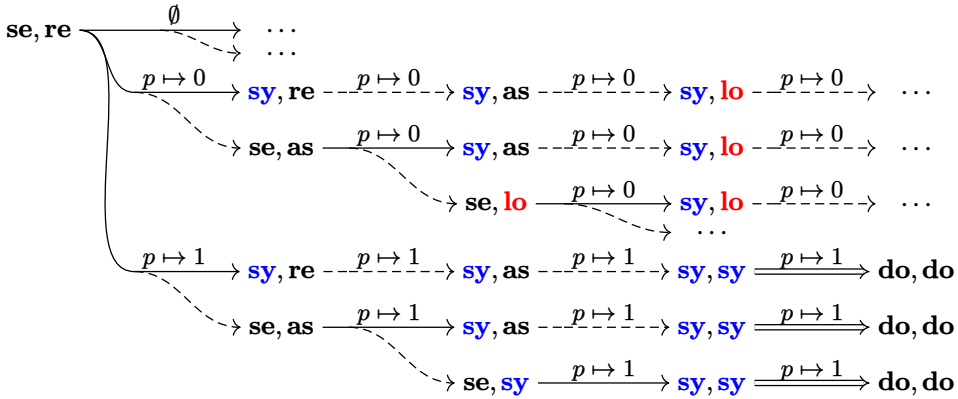


Figure 5.4: This shows the (search) tree of constructive rounds starting from the workers initialised from the protocol  $\{(\mathbf{send} \{0, 1\} \hookrightarrow p; \mathbf{sync}), (\mathbf{recv} p \hookrightarrow v; \mathbf{assert} v; \mathbf{sync})\}$ , from Example 5.5. Each worker is identified by the first two letters of its statement, e.g., worker  $\langle\{v \mapsto 0\}, (\mathbf{assert} v; \mathbf{sync})\rangle$  is abbreviated as ‘as’. We colour **lo** (for **loop done**) and **sy** to reveal patterns. Arrows depict (a)synchronous steps as usual, but we *dash* them iff only their *second* worker is updated. Looping traces are truncated to  $(\dots)$ .

**Example 5.6** (Examples of Protocols Constructing Behaviours).

- $\{\mathbf{loop} \mathbf{sync}\}$  constructs arbitrarily many rounds without any messages. Again, this primitive protocol behaves like the trivial, empty protocol  $\{\}$ .
- $\{\mathbf{recv} p \hookrightarrow v; \mathbf{sync}\}$  is stuck awaiting a message that is never sent.
- $\{\mathbf{recv} p \hookrightarrow v; \mathbf{send} \{v\} \hookrightarrow p; \mathbf{sync}\}$  has a cyclic dependency on the message at port  $p$ . The **recv** blocks forever for a message that is never sent.
- $\{(\mathbf{send} \{0, 1\} \hookrightarrow p; \mathbf{sync}), (\mathbf{recv} p \hookrightarrow v; \mathbf{send} \{6+v\} \hookrightarrow p'; \mathbf{sync})\}$  demonstrates the propagation of a nondeterministic choice between workers, through a message at port  $p$ . The recipient is oblivious to the nondeterminism, expressing the functional relationship  $m(p) + 6 = m(p')$ .

Because constructed rounds are growing, many prior results also apply to construction. Rounds are still determined by their behaviour (Theorem 5.1), and composing protocols preserves their constructed behaviours (Theorem 5.2, because the new round is constructive if both given rounds are constructive). However, unlike acceptance, construction is not *decompositional* (Theorem 5.3): composite protocols construct behaviours *not* constructed by the constituents independently. Example 5.7 demonstrates behaviour emerging from protocol interactions.

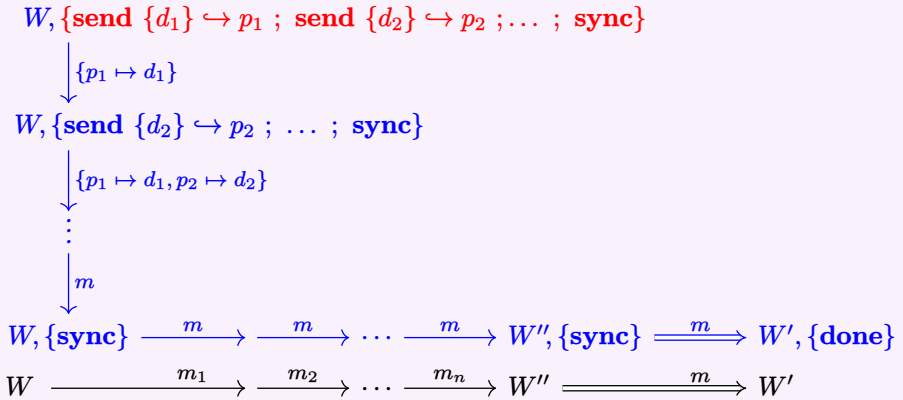
**Example 5.7** (behaviour constructed from protocol interactions). Consider primitive protocol  $\{(\mathbf{send} \{1\} \hookrightarrow p_1; \mathbf{recv} p_2 \hookrightarrow v; \mathbf{sync})\}$  where  $p_1 \neq p_2$ , and the same but with  $p_1$  and  $p_2$  swapped. Independently, each protocol forever awaits a message that

is never offered. But their composition constructs  $[\{p_1 \mapsto 1, p_2 \mapsto 1\}]$ , where each worker first sends and then receives the message sent by its peer.

Intuitively, construction resembles (nondeterministic) *actor programming*: computation and communication unfold inter-dependently. Theorem 5.4 precisely relates the behaviours accepted and constructed by each protocol  $S$ : behaviours accepted by  $S$  are those constructed by  $S$  in composition with some existentially quantified ‘oracle’ protocol, which stands in for the environment of  $S$ , offering whatever messages  $S$  needs to end the round. Like accepted behaviours, a protocol’s oracles are generally not enumerable unless ports, messages, or behaviours are finitely enumerable. This gives PDL programmers another view on acceptance: the behaviours their protocol can construct as a part of any (composite) protocol.

**Lemma 5.4** (round oracle). Given growing round  $W \xrightarrow{*} \xrightarrow{m} W'$ , there exists oracle  $s_o : \mathcal{S}$ , worker  $w'_o$ , and constructive round  $W \cup \{\langle \emptyset, s_o \rangle\} \xrightarrow{*} \xrightarrow{m} W' \cup \{w'_o\}$ .

*Proof.* Build an oracle (in red) that sends each message in  $m$  in some order before a **sync** statement. Build a round (in blue) in two stages. The first (vertical) stage builds  $m$ , one message at a time, while  $W$  takes no steps. The second (horizontal) stage reproduces the given round (in black), but made  $m$ -constant with Lemma 5.2.



□

**Theorem 5.4** (acceptance vs. construction). Protocol  $S$  accepts behaviour  $M$  iff  $(\Leftrightarrow)$  there exists an ‘oracle’ protocol  $S_o$  where  $S \cup S_o$  constructs  $M$ .

*Proof.*  $(\Rightarrow)$  By Lemma 5.2, a trace of constant rounds  $r$  exists whose behaviour  $M \triangleq [m_1, m_2, \dots, m_n]$  is accepted by  $S$ . Apply Lemma 5.4, but generalized to an  $n$ -round oracle  $(o_1 ; o_2 ; \dots ; o_n)$  constructing  $M$ .  $(\Leftarrow)$  By definition, a trace of constructive rounds  $r$  with behaviour  $M$  from  $S \cup S_o$  exists. By Definition 5.15,  $S \cup S_o$  accepts  $M$ . Finally, by Theorem 5.3,  $S$  accepts  $M$ , *i.e.*, removing  $S_o$  preserves acceptance. □

Acceptance and construction provide useful, dual views on behaviour. Their differences reflect their different assumptions about the execution environment. Acceptance reflects ignorance of the environment. This view is useful to users, who reason about the behaviour of known protocols as parts of unknown composite (session) protocols. Construction is holistic, in the sense that it closes the system, only considering messages as they are sent, interleaved with worker updates. This view is appropriate for reasoning about systems at large, which users will do some of the time, but which PDL Runtimes do all of the time.

## 5.4 PDL Runtimes

Section 5.3 defined the PDL syntax and semantics. In these terms, we specify and demonstrate the user interface and correctness properties of PDL Runtimes.

### 5.4.1 A Specification of PDL Runtimes

A PDL Runtime is an interactive system that realises a communication session. Each runtime configuration  $c, c_1, c', \dots : \mathcal{C}$  has a *current* (session) protocol. Users communicate indirectly, by directly and arbitrarily interleaving user actions:

1. Some user *injects* a chosen input protocol into the current protocol.
2. Users observe the behaviour that results from *running* the system.

Precisely, a PDL Runtime instantiates the signature in Definition 5.17, *i.e.*, fixes the *configuration* type  $\mathcal{C}$ , and then gives sound definitions of the four user-facing operators on runtime configurations.

**Definition 5.17** (PDL Runtime signature).

$$\mathcal{C} : \text{Type} \quad \text{start} : \mathcal{C} \quad \text{run} : \mathcal{C} \rightarrow \mathcal{M}^* \times \mathcal{C} \quad \text{proto} : \mathcal{C} \rightarrow 2^{\mathcal{S}} \quad \text{inject} : \mathcal{C} \rightarrow 2^{\mathcal{S}} \rightarrow \mathcal{C}$$

where, as introduced earlier,  $\mathcal{S}$  is the domain of PDL programs,  $\mathcal{M}$  is the domain of message maps, and  $\mathcal{M}^*$  is the domain of message map lists, *i.e.*, behaviours.

The PDL Runtime preserves the users' basic expectations of the current protocol by preserving Properties 5.1 and 5.2.

**Property 5.1** (initially trivial protocol).  $\text{proto}(\text{start}) = \{\}$ .

**Property 5.2** (injection composes).  $\text{proto}(\text{inject}(c, S : 2^{\mathcal{S}})) = \text{proto}(c) \cup S$ .

Additionally, PDL Runtimes must ultimately preserve each of the *runtime invariants* defined in Section 5.2. Each behaviour must be observed consistently between users (Invariant 5.1), accepted by each component of the current protocol (Invariant 5.2), the result of a round if it exists (Invariant 5.4), and such that each message must have been sent by a user-injected protocol (Invariant 5.3).

Because constructed behaviours are accepted by each injected part of the session protocol, it suffices for the PDL Runtime to preserve Properties 5.3 and 5.4.

**Property 5.3** (soundness). Each  $run(c) = \langle M, c' \rangle$  implies  $proto(c)$  constructs  $M$ .

**Property 5.4** (completeness). For all  $c : \mathcal{C}$ , if any  $M \neq []$  exists that is constructed by  $proto(c)$ , after some finite run-steps from  $c$ , any  $M' \neq []$  is observed.

Note that Property 5.4 leaves unspecified *which* non-empty behaviour  $M'$  is constructed from  $c$ , leaving room for non-deterministic execution.

## 5.4.2 Example: The Silent PDL Runtime

To demonstrate our PDL Runtime specification, we consider the *silent PDL Runtime* (Definition 5.18), which reuses composite protocols ( $2^S$ ) as configurations ( $\mathcal{C}$ ), and instantiates the operators comprising the PDL Runtime signature (Definition 5.17).

**Definition 5.18** (silent PDL Runtime).

$$\mathcal{C} \triangleq 2^S \quad start \triangleq \{\} \quad run(S) \triangleq \langle [], S \rangle \quad proto(S) \triangleq S \quad inject(S, S') \triangleq S \cup S'$$

Intuitively, the silent PDL Runtime accurately manages the session protocol, but it does nothing else. It is easy to see why this implementation satisfies Properties 5.1 to 5.3: it simply collects injected protocols, and always produces zero-round behaviour  $[\ ]$ , which every protocol constructs, by definition. Our artefact includes a simple proof of each of these properties. But clearly this implementation is not *complete* (Property 5.4). We use protocol  $\{\}$  constructing  $\{\} \xrightarrow{\emptyset} \{\}$  as witnesses for our proof of *incompleteness*: while  $\{\}$  constructs  $[\emptyset]$ , run steps only ever observe  $[\ ]$ .

Hopefully, this demonstrates that the preservation of completeness is the first major challenge in implementing correct PDL Runtimes. The difficulty arises from the fact that rounds may complete after arbitrarily many (finite) steps. Indeed, it is generally undecidable whether given workers can synchronise; it is analogous to the halting problem. Fortunately, because finite protocols offer finitely many messages, the tree of traces rooted at any finite workers  $W$  is finite *per depth*. Figure 5.4 visualises how offers and updates guide these searches. Of course, typical search optimisations can improve efficiency drastically. For example, by caching visited workers, many *transpositions* (the same workers reached via distinct traces) are avoided. We expect the properties of PDL to afford even more optimisations. For example, because of *determinism* (Theorem 5.1), we expect completeness to be preserved if  $W \xrightarrow{m[p:=*]} W'$  stays unexplored once a constructive  $W \xrightarrow{m[p:=d]} W'$  is discovered. But presently, rigorous proofs of these claims are still future work.

## 5.5 Future Work

### 5.5.1 Formalising the Rest of the Connector Runtime

We strive to recreate the existing Rust implementation of the Connector Runtime as a PDL Runtime, atop our present specification, such that we can check its correctness. Here, we overview its facets which remain to be formalised.

**Round Search** We conjecture that it is decidable whether given workers synchronise after a bounded number of asynchronous ( $\rightarrow$ ) steps. Such a proof lays the groundwork for terminating search algorithms that are correct up to a *fuel* :  $\mathbb{N}$  bound. For example, we imagine modelling the *timeout* mechanism of the Connector Runtime as run fuel. Many round-search algorithms are conceivable. Which of them are correct and efficient? We also hope to model the algorithm underlying the Connector Runtime and (ideally) verify its correctness.

During the formalisation of the semantics of PDL, we also discovered an elegant relation between the description of a protocol as a proposition and the runtime implementation of a protocol as a proof. In proof terms, a protocol states the existence of a round (or an  $n$ -round trace), and the runtime works to find a proof of that proposition. We aim to explore this relation further in the future, to discriminate runtime algorithms from a proof theoretic perspective.

**Distributed Workers** Distributed PDL Runtimes partition the workers in each configuration  $\mathcal{C}$  over a network of *processes*  $\pi, \pi', \pi_1, \dots : \Pi$ . Processes only interact by passing asynchronous *control messages*. The processes cooperate to simulate a search (see *Round Search*, above). The execution of all workers sharing (data) messages is simulated: each process  $\pi$  isolates its subset of workers in a local message environment  $m$ , and then  $\pi$  explicitly informs its peers of newly-sent message  $\langle p, d \rangle$  via a control message  $\langle \text{sent}, n, m, p, d \rangle$ , where  $n : \mathbb{N}$  identifies the current round, to avoid confusion if control messages arrive late.

The processes realise a synchronised round  $W \xrightarrow{*} \xrightarrow{m} W'$  through explicit coordination. Each round, an elected process selects and announces an arbitrarily chosen  $m$  after learning that, for each process  $\pi$  (including itself), there exists some  $m_\pi \subseteq m$ , where, locally,  $W_\pi \xrightarrow{m} \xrightarrow{m} W'_\pi$ . This information is communicated to the leader from  $\pi$  via the control message  $\langle \text{ready}, n, m_\pi \rangle$ . To minimise the burden on the leader, the Connector Runtime lets processes aggregate this information toward the leader. A sink tree is overlaid atop the network, with the leader at the root. With  $\langle \text{ready}, n, m \rangle$ , a child process  $\pi$  informs its parent that each process in the subtree rooted at  $\pi$  is ready to synchronise with some  $m_\pi \subseteq m$ .

**Restricted Ports** Users of the Connector Runtime define *access* as a process-port relation, such that each process  $\pi$  can send and receive messages at port  $p$  only if  $access(\pi, p)$ . Port messages are only ever observed by accessors. For example,  $\langle sent, n, m, p, d \rangle$  is sent only to  $p$ -accessors, and  $m$  omits messages of ports not accessed by the recipient. Users of the Connector Runtime also define which process *determines*:  $\mathcal{P}_d \rightarrow \Pi$  each *deterministic* port  $\mathcal{P}_d \subseteq \mathcal{P}$ . Whenever  $\pi = determines(p)$ , only workers at  $\pi$  may execute **send**  $E \hookrightarrow p$  where  $|E| \leq 1$ , *i.e.*, the nondeterministic choice is trivial. The benefit of deterministic ports is that their values can be omitted from control communications while preserving determinism: all growing rounds that start from the same  $W$  with the same messages at ports  $\mathcal{P} \setminus \mathcal{P}_d$  necessarily have the same messages at port  $\mathcal{P}_d$ .

Access and determinism give users greater control over how the PDL Runtime communicates port data via control messages. For example, users can restrict the observation of messages at particular ports to (the users at) particular processes.

**Channel API** The Connector Runtime lets users express *access* and *determines* (see *Restricted Ports*, above) implicitly via its API, which is a restriction of that described in Section 5.2: sets of users rendezvous at sets of IP addresses; each rendezvous includes only a *sender* and *receiver* process, and creates a new *channel* comprised of ports  $p$  and  $p_d$ . The latter is accessed only by this sender and receiver and is determined by the sender. Users understand  $p_d$  as carrying user data across the channel from the sender to the receiver only, while  $p$  is used in session-wide control communications, *e.g.*, to discriminate values sent at  $p_d$ .

As users never read values at  $p$ , these act purely as discriminators and can be chosen arbitrarily by the Connector Runtime. In fact, it suffices if the values at  $p$  are restricted to  $\{0, 1\}$ , such that control messages can be densely packed into bit vectors. Each  $p$  is split into multiple ports if more bits are necessary.

## 5.5.2 Exploring Variations of the PDL

We intentionally minimise the coupling between the syntax and semantics of PDL primitives (Section 5.3.1) and the rest of the language (Sections 5.3.3 and 5.3.4) to ease future experimentation with variants of PDL.

**Ports as Data** If PDL is changed such that  $\mathcal{P} \subseteq \mathcal{D}$  (ports are data), protocols become significantly more expressive and flexible. For example, **send**  $\{e_d\} \hookrightarrow e_p$  sends dynamic data ( $e_d$ ) at a dynamic port ( $e_p$ ). However, this would make protocols more difficult to reason about; *e.g.*, generally, we could not statically check which workers or processes *access* given ports (see *Restricted Ports*, above).

**N-Round Lookahead** We consider definitions of *productivity* (Invariant 5.4) which change the role of synchronisation. For example, we consider a generalisation where PDL Runtimes must always *maximise* the number of completed rounds up to a *lookahead* bound which is currently implicitly 1. Consider Example 5.8, which synchronises  $N$  rounds before blocking forever, where  $N$  is the data at port  $p_N$  in the first round. With 1-lookahead, the PDL Runtime is correct to complete the first round with behaviour  $\{p_N \mapsto 1\}$ . But with 3-lookahead, this would be incorrect, because it would not produce the maximal number of rounds up to the end of round 3, *e.g.*, where  $\{p_N \mapsto 3\}$  and  $\{p_N \mapsto 182\}$  are maximal instead.

**Example 5.8** (*N-Sync*). `recv  $p_N \hookrightarrow v$  ; while  $v$  do (write  $v - 1 \hookrightarrow v$  ; sync).`

**Local Synchronisation** We consider decoupling different workers' synchronisations, *e.g.*, as in the Reo semantics. Precisely, an unspecified subset of workers *stutter* each round: maintaining their states and not interacting. Intuitively, this change empowers the PDL Runtime to make more decisions at the cost of weakening the meaning of **sync**. In practice, this change prevents slow workers from impeding productivity, but it introduces threats of unfairness and starvation, unless they are prevented, *e.g.*, by users specifying how workers are prioritised. This change affords the concurrent synchronisation of local *synchronous regions* of the system, as is done with Reo in [Arb11], and with Dreams in [PCdVA12].

### 5.5.3 Definition, Analysis, and Optimisation of PDL Protocols

PDL is designed to afford (de)compositional reasoning about protocols and their properties. We see benefit in future work that develops a corpus of PDL protocols that solve useful problems, or have desirable properties.

We are particularly interested in letting PDL Runtimes transparently optimise execution by recognising protocols and exploiting their known properties. Most interestingly, these opportunities may be unknowable to the users, because users lack the overview of the session protocol, *e.g.*, in ad-hoc sessions between strangers over the Internet. For example, if Amy in Helsinki routes outgoing messages to Bob in Tokyo, and Bob filters incoming messages before forwarding them to Dan in Helsinki, the runtime can transparently re-configure the session to filter Amy's messages before routing them to Dan, *i.e.*, avoiding the inter-continental round-trip. Such session optimisations are already supported by (and benchmarked with) the Connector Runtime [EH24], but currently, they must be recognised and triggered manually. The Reewolf project documentation [EH24] highlights the potential of systematic protocol analysis and transformation via existing tools. For example, can we encode PDL protocols as annotated port-graphs and protocol transformations as graph-rewriting rules, *e.g.*, using the PBPO<sup>+</sup> graph-rewriting formalism and tools [OER23]?

## 5.6 Related Work

### 5.6.1 Related to Behavioural Specification with the Reowolf PDL

Here, we compare PDL to well-researched synchronous languages Reo, Esterel, and Lustre. We leave out of scope the comparisons to the many other (synchronous) languages such as Signal [GLGB87,BLGJ91], ARx [PC20], and HipHop.js [BS20].

**Reo** PDL is directly inspired by Reo, so the languages have many similarities. Reo is also used to specify and coordinate synchronous and multi-party communications [Arb04,Arb11] in interactive systems, regulating the interactions between software components as constraints on their messages. Notably, the compiler [DLA<sup>+</sup>18] of textual Reo [DA18] with constraint automata semantics [BSAR06] generates executable *coordinators* that interface with users. Reo is more declarative<sup>2</sup>, easing static protocol verification and transformation, and top-down coordinator re-configuration [KGV13,KCPA08,KMLA11,Kra11].

However, there is less emphasis on the dynamic composition of Reo protocols, or their application to distributed systems. For example, Reo coordinators are centralised, and cannot interface with other coordinators. In contrast, Reowolf is designed around its dynamic and distributed applications: the protocol unfolds dynamically as users inject new protocol components on the fly.

**Esterel** PDL and Esterel have similar syntax, and the same control flow: sequential, but punctuated by synchronous message passing [BG92,Edw02b,FYTF19].

Esterel reflects its design for a different use case: *real-time reactive systems*, which emphasise timely reactions to external events. Hence, notably, Esterel and PDL have very different relationships with nondeterminism: both embrace it for program composition, but only in PDL are nondeterministic programs executable. Esterel programmers are responsible for ensuring determinism, whereas PDL programmers rely on the behaviour being determined at runtime. In fact, nondeterminism is desirable in PDL protocols, because it affords flexibility to later protocol composition and refinement. Effectively, PDL programmers enjoy ignorance of their environment, at the cost of complexity in the runtime system.

**Lustre** Lustre [HCRP91,Hal05] semantically separates synchronous computations (within a logical instant) and sequences of computation (over consecutive instants). Lustre has operators calculating (arithmetic) functions and operators ‘scheduling’ functions over time. Composition of synchronous programs is done per instant, on

---

<sup>2</sup>Most of the imperative (‘less declarative’) nature of PDL is a consequence of its statement syntax (Figure 5.2) and the memory-mutating small-step update semantics (Figure 5.3). But a major motivation for exploring PDL variants (Section 5.5.2) is to recapture some of the declarative feel of Reo. How far can we go while preserving the present expressivity and key properties of PDL?

a shared memory machine. Lustre provides some meaningful primitives to specify these functions constructively, which requires them to be deterministic.

In comparison, PDL has nondeterministic and synchronous primitives. Agents interpret PDL programs and interact with other agents. Agents explicitly delineate instants with the **sync** statement, whose semantics guarantees the consistent values at variables shared between agents (*i.e.*, ports).

## 5.6.2 Related to Network Programming with Reowolf Connectors

**Dreams** Like the Connector Runtime, the engine of the Dreams framework [PCdVA12] translates Reo-like synchronous protocols into networks of automated, message-passing agents, coordinated via distributed consensus algorithms. The works focus on different features. Dreams statically partitions the system into *synchronous regions*, enabling (only) region-local synchronisation and reconfiguration. In contrast, Reowolf has global synchronisation (*i.e.*, the trivial case of Dreams), but lets users change the protocol on the fly. We see the best features of Dreams and Reowolf as complementary, and we see promise in combining their strengths, *e.g.*, by relaxing the meaning of our **sync** statement, as discussed in Section 5.5.2. Can future sessions have synchronous regions (as in Dreams), while allowing users to change protocols in synchronous regions on the fly (as in Reowolf)?

**Bulk Synchronous Processing (BSP)** BSP is an abstract model of parallel computation [Val90]. The main idea is to break time into sequential *supersteps*, within which, processors compute in parallel, and between which, processors pass messages and synchronisation barriers. Literature has developed software and hardware for BSP, typically for the purpose of high performance computing, maximising program portability [CFSV95, GV94] and performance [dRRdQGR<sup>+</sup>15, WBG23, ZPA<sup>+</sup>21]. Many BSP ideas are also evident in Reo(wolf), *e.g.*, in emphasising the separation of computation from communication. Reowolf takes it further: like BSP supersteps, rounds of PDL communication behaviour are separated by synchronisation barriers. Which BSP hardware / runtime environments are suitable PDL Runtimes?

**Message Passing Interface (MPI)** The MPI standard [CGH94] eases and decouples tasks of 1. library developers of general-purpose languages, and 2. their users implementing data-parallelism in general, and large simulations in particular [Gro11].

Reowolf and MPI have in common that they augment host applications with high-level abstractions for coordinated, multiparty message passing. MPI-2 in particular overlaps with Reowolf in expressing task-parallelism via workers.

While Reowolf and MPI differ considerably in the details of their programming abstraction, more fundamentally, they have different consequences on runtime beha-

viour. Reowolf focuses on internet programming, so PDL affords a distributed and continuous refinement of the session protocol. In contrast, MPI affords the abstract specification of task- and data-parallelism in programs at compile-time, such that programmers can reason about run-time performance.

**Software Defined Networks (SDN)** Like Reowolf, SDNs [HHB14] provide an abstraction over distributed systems. For example, the OpenFlow [LKR13] control protocol is used for remote administration of network switch packet-forwarding tables. Rules can be made on a controller, and dynamically pushed to the network switches, *e.g.*, to change the routing algorithm [MAB<sup>+</sup>08]. Tools such as NorthStar [Kra18] and Khathará [BILB18] deploy and manage containerised applications atop SDNs.

SDNs and Reowolf provide abstractions over different layers in the OSI stack. SDNs are suited to network administrators, while PDL isolates coordination logic typical to applications. As such, we see potential for their combination. Can connectors efficiently route messages (via transparent usage of SDN technologies) within the bounds of what users accept (as users specify via PDL protocols)?

**Multi Party Session Types (MPST)** Dyadic session types specified the communication behaviour of message channels [Hon93]. MPSTs generalised these to multiparty sessions; each *global* MPST (specifying the session at large) is projected onto each peer, yielding *local* MPSTs (specifying roles in the session) [HYC08]. MPST variants are embedded in host languages such as Rust [CBT22], Scala3 [CEJP22], Haskell [LM16], and OCaml [ILN22], reducing the verification of (dynamic) session properties, such as deadlock freedom, to (static) type-checking of host programs.

Both MPST and Reowolf's PDL specify communications via nondeterministic, message-passing programs. Both paradigms also let (control) messages keep peers' nondeterministic choices aligned: senders choose and recipients follow. But where MPSTs usually specify asynchronous messaging, and are statically decomposed from a global specification, PDL protocols specify synchronous messaging, and are dynamically composed throughout the session. For example, [Ney13] dynamically coordinates (untyped) Python programs, and [BY08] adopts synchronous coordination. We expect MPST to inspire applications of PDL in the future, particularly for common uses of MPST: static program analysis, generation, and transformation.

**Publish / Subscribe Protocols** The publish/subscribe paradigm characterises a family of decentralised network protocols (such as XMPP [HW10] or MQTT [SM17]) intended for lightweight communications between IoT devices. For instance, in MQTT, users can (un)subscribe to topics, or publish data to a topic, which persistent, distributed, automated *broker* agents disseminate to subscribers. Likewise, PDL Runtimes coordinate user communications. However, because PDL specifies the

synchronous inter-dependencies of messages, PDL affords users specifying the ordering and data inter-dependencies between multiple parties' messages; *e.g.*, PDL affords the definition of distributed transactions.

## 5.7 Conclusion

We have contributed a formalisation of the essential contributions of the Reowolf project, and demonstrated the promise of Reowolf connectors as a network API for multi-party internet applications. Like the classic BSD-style sockets, connectors pass messages between users over the network. Unlike sockets, users control session behaviour by continuously refining the session protocol, which is expressed in Reowolf's Protocol Description Language (PDL). Consequently, two concepts coincide in PDL protocols: (1) user specifications of the session behaviour, and (2) executable programs delegated by users to the runtime system.

(Article [ELHA25] underlying) this chapter contributes formal definitions of (key properties of) PDL, specifies (key requirements on) the implementation of the Connector Runtime, and explains their connection. We show how PDL's dual semantic notions of protocol behaviour support the complex requirements on connectors. PDL is declarative, as it lets users (de)compositionally express and reason about the behaviour *accepted* by (their own) parts of the session protocol. But PDL is also imperative, as PDL-runtime systems can *construct* behaviour one synchronous round at a time, by executing the session protocol, throughout its refinement by the users.

These contributions lay the groundwork for future work to (re)define particular PDL protocols and runtimes, and to rigorously formalise and prove their properties. We identified particularly promising future directions. For example, to verify the correctness of the Connector Runtime, we must first verify the correctness of its underlying round-search algorithm. These efforts contribute to the greater vision of extending the rigour and programmability of formal protocol languages to the decentralised and dynamic world of internet programming.

**General Takeaways** This chapter preserves the main idea in prior chapters: agents cooperate by incrementally refining a shared specification, by extending it with new parts. But now this idea is applied in a new context, where the agents and their runtime system are distributed over the network. Formalising PDL clarified the properties that let PDL protocols act as the interface between the agents and their automated runtime system. Notably, Theorem 5.2 ensures that agents' contributions preserve their independent specification of accepted behaviour. And Theorem 5.4 ensures that the automatic execution of the shared protocol is feasible, and the resulting behaviour is accepted by construction.

# JustAct: A Framework for Policy-Regulated Multi-Domain Data Processing

## Abstract

Inter-organisational data exchange is regulated by norms originating from sources including (inter)national laws, processing agreements, and individual consent. Verifying norm compliance is complex because laws distribute responsibility and require accountability. Moreover, in some application domains (*e.g.*, healthcare), privacy requirements extend the norms (*e.g.*, patient consent). In contrast, existing solutions such as smart contracts, access- and usage-control assume policies to be public, or otherwise, statically partition policy information at the cost of accountability and flexibility. Instead, our framework prescribes how decentralised agents *justify* their actions with policy fragments that the agents autonomously create, gossip, and assemble. Crucially, the permission of actions is always reproducible by any observer, even with a partial view of all the dynamic policies. Consequently, actors know that future auditors will confirm their permissions. Systems centralise control by (re)configuring externally synchronised *agreements*, the bases of all justifications. This centralises control only to the extent desired by the agents.

We define the JustAct framework, detail its implementation in a particular data-processing system, and design a suitable policy language based on logic programming. A case study reproduces BRANE – an existing policy-regulated, inter-domain, medical data processing system – and serves to demonstrate and assess the qualities of the framework. We show how JustAct improves BRANE by enabling the amendment of data sharing agreements at runtime, while clarifying how to reason about and conduct audits on data consumers and processors.

**Basis of this Chapter** This chapter is primarily based on the article [EMvB25a], which is under review for publication in the LMCS2025 journal, and which extends the FORTE2024 conference paper [EMvB24] with the new BRANE system case study, which is based on the BRANE section of the FGCS2024 journal article [KME<sup>+</sup>24] and a paper at (the ReWorDS workshop at) the eScience2022 conference [EMvBB22].

## 6.1 Introduction

Data exchange systems are distributed systems facilitating the controlled sharing, trading, and processing of (often large) datasets and analysis results within data exchange applications, increasing the public, commercial, or academic value of collected data. Following the inter-organisational nature of data exchange systems and the (market- or privacy-)sensitive nature of the exchanged assets, collaborating organisations adopt complex governance models [TGMD22] in an attempt to ensure compliance with regulations and contractual agreements. In support of such governance models, high levels of control should be given to organisations to influence the execution of data exchange applications, *e.g.*, via access control [QTD<sup>+</sup>20, SdV00] or usage control [JD22, ZPSP05, MLP<sup>+</sup>19]. Furthermore, high levels of accountability are required to support dispute resolution [SMV<sup>+</sup>19] and to demonstrate legal compliance [CT22]. Data exchange systems exhibit a fundamental trade-off between maximising the availability of data to data users and maximising the control over data to data owners, subjects, and (privacy) authorities. In this work, we present a framework that enables the organisations collaborating in a data exchange system to formalise their shared and individual positions with respect to this trade-off through declarative *policies*.

Our approach is to define a framework for runtime systems that specify the interaction between system dynamics (messages and actions) and statics (policies and facts). In one direction: agent messages create and disseminate policies. In the other direction: policies specify which actions are permitted. Actors are held responsible for collecting and providing valid *justifications* of their actions: sufficient policy information to prove permission. Crucially, permission is decidable, despite the agents having only partial knowledge of the existing policies, as policies can be changed dynamically. Actors can thus be confident that other actors (*e.g.*, auditors) will agree that their actions were permitted. Figure 6.1 visualises this process; human users and automated services reason about and influence the behaviour of their peers by creating and sharing policies. The framework standardises the meaning of well-behavedness: act only as permitted. However, its *enforcement* is an orthogonal matter; implementations of the framework can prevent non-permitted actions (*ex-ante* enforcement), recognise and punish misbehaviour after it has happened (*ex-post*

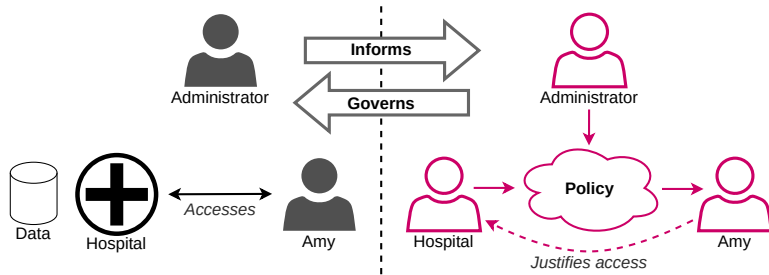


Figure 6.1: Conceptual use-case of the framework. Agents autonomously choose to act, accessing data in the real world (left), as permitted by shared policies (right).

enforcement), by a centralised authority, in a peer-to-peer fashion, or whatever mixture is appropriate. As such, our framework is flexible to the system implementation whilst ensuring large degrees of control for agents and their mutual agreements.

Agents express and communicate consistently by using a shared *policy language*, which is fixed by each instantiation of the framework. However, different languages are available, and their characteristics determine the power and complexity of policies. In this chapter, we characterise the space of suitable languages in general, and demonstrate two languages in particular. In the earlier sections, we use *Datalog*<sup>-</sup> to easily show key concepts. Later, we define Slick as a *Datalog*<sup>-</sup> variant, and use it to express more realistically complex policies. To minimise the burden on the reader, both languages are based on the intuitive and ubiquitous *Datalog* language, and are sufficiently characterised within the chapter.

This work reports on the ongoing developments of generic, policy-driven data exchange systems that satisfy legal requirements. We intend to make our approach an integral part of the EPI framework [KME<sup>+</sup>24], to improve its accountability, auditability, and preservation of user privacy. This chapter reflects our latest steps in this direction. We implement a generic data exchange system around our framework. Then we conduct a case study, detailing and evaluating the application of our implementation to new and existing usage scenarios of the BRANE system, which is the component of the EPI framework responsible for coordinating and regulating the execution of medical data processing workflows. Precisely, after we summarise some background literature (Section 6.2), we contribute:

1. a **framework** for multi-agent runtime systems in which agent actions are regulated by policies that agents autonomously create and share (Section 6.3),
2. an **implementation** of a generic data exchange system: agents create and share policies in a configured policy language to regulate data access (Section 6.4),

3. definition of the Slick **policy language** and interpreter, for expressing policies with realist abstractions and relationships (Section 6.5), and
4. a **case study**, evaluating our framework, instantiated for data exchange with Slick policies, in application to BRANE’s medical workflow processing (Section 6.6).

We discuss our contributions by their own merits (Section 6.7) and in comparison to related work (Section 6.8) before we conclude with a summary (Section 6.9).

Note that the source code for our implementations and experiments are included in the accompanying artefact, available at [EMvB25a], including the Rust implementations of the framework as an ontology, the data exchange runtime system, the runtime-trace analyser tool, the Slick interpreter, the initialisation of the prototype for BRANE, and the unit tests implementing our experiments.

## 6.2 Background

### 6.2.1 Distributed Systems and Algorithms

*Distributed systems* model the distribution of a stateful *configuration* over *processes*; each process has its own local *state*. *Distributed algorithms*, implemented by each process, give systems useful emergent properties. Often, these algorithms are defined in terms of only basic message-passing primitives, as in IP and UDP protocols: messages are asynchronous, and message delivery is unreliable. Some algorithms solve complex distributed problems (*e.g.*, self-stabilisation), and create useful abstractions over the distributed system (*e.g.*, synchronisers).

We refer to two classes of algorithms. Firstly, *gossip* protocols disseminate information by replicating and forwarding messages from peer (process) to peer, resulting in decentralisation and robustness, by imposing minimal requirements on the network topology and process behaviour [BCFH08]. Secondly, *consensus algorithms* establish fundamental agreement on the selection of a particular value, consistently among processes. Consensus has been well-studied for decades [RBA05], but has seen renewed interest in application to blockchain technologies in, for example [KdHH22, KT22].

### 6.2.2 Multi-Agent Systems and Autonomy

The field of *multi-agent systems* studies processes (called *agents*) that exhibit social phenomena as a result of their *autonomy*: agents are motivated by goals to draw from their partial information to act on shared resources and interact with other agents. The literature explores the problems of distributed systems, but also explores a variety

of (software models of) social organisations, ranging from cooperative data-sharing *consortia* (e.g., in [Fer23]) to competitive markets (e.g., in [ZBL<sup>+</sup>23]).

*Agent-oriented programming* spreads program logic over agents, balancing the concerns of software (language) engineering and object-oriented programming, but with a unique emphasis on agent autonomy, e.g., to improve system scalability and robustness. These ideas are present in seminal works such as [Sho93] and persist into more recent works such as [MWY17].

### 6.2.3 Policy Specification Languages

We give a brief overview of the various notions of ‘policy’ which were developed by various disciplines and which influenced our work.

*Access control* is a mainstay in cyber-physical systems (e.g., from databases to IoT networks) that revolves around the regulation of agents accessing resources. For example, a policy specifies in which case a given agent is *authorised* to read or write a given data asset. Policies often take the form of conditional rules [San98, FJT22], sometimes applied in the context of meta-data attributes [SO17]. *Usage control* generalises access control to the control of usage events that occur for the *durations* of time the resource is being used. Usage authorisation must be maintained and can be interrupted [AK22] which may be realised with continuous monitoring [HIA<sup>+</sup>23] and mutability of attributes [SP03]. XACML [ANP<sup>+</sup>03] and ODRL [Ian07] are popular, standardised policy languages for expressing access and usage conditions as policies. For example, [UMSB12] implements usage control in XACML.

*Normative specifications* define fundamental normative positions (relations) such as powers, duties, rights, obligations, and permissions [AGNvdT13] between actors. Normative specifications can formalise (and make machine-readable) the norms described in normative documents such as laws and regulations, organisational policies, and contracts. Laws and regulations are specified to apply to activities of particular kinds within particular jurisdictions. For example, the European Union’s General Data Protection Regulation (GDPR) [Eur16] regulates the processing of personal data within the European Union, but its wide reach and impact makes it influential even outside the EU.

The study of norms reflects its long history in its rich nomenclature, for example, [BvdT08] clarifies the relationship between *substantive* and *procedural* norms, and [GIM<sup>+</sup>18] details how open terms in norms intentionally leave room for interpretation, e.g., by a judge during the resolution of a particular dispute. Normative specification languages attempt to capture this nomenclature to enable the encoding of norms such that they can be used as enforceable policies within software systems.

A wealth of literature connects the aforementioned notions of policy, with the express goal of clarifying their interrelations and regulating the behaviour of (actors within) software systems. For example, the eFLINT language [vBLvDvE20] formalises norms using the Hohfeldian framework of legal proceedings [Wes13] and has been used for access control [vBKB<sup>+</sup>21]. Symboleo [SPA<sup>+</sup>20] is also based on Hohfeld’s framework but, contrary to eFLINT, is focussed on contract specifications. FIEVeL [VC07] is a language used to verify properties of (social) institutions. These languages afford the application of various tools and techniques to policies. For example, model-driven development [S<sup>+</sup>06] and model-checking for high-level properties in policies of both FIEVeL [VC07] and Symboleo [PRR<sup>+</sup>24].

In [EvB24] (Chapter 4), we explored *cooperative specification*, a process in which multiple agents develop a shared (policy) specification together, taking turns contributing to the policy. Crucially, the ongoing cooperation is regulated by a meta-level agreement between the agents constituting when a policy is considered *valid*. Suitable policy languages must define this semantic notion of validity, giving the agents meaningful control over the policy being developed. For example, Amy formalises ‘Only Amy can access Amy’s data’ which would be invalidated by Bob’s contribution of ‘Bob accesses Amy’s data’; Amy’s contribution captured a meaningful constraint on Bob’s contribution, and thus also on the developed policy. In the extreme case, the lines blur between the specification of what events occur in the system, what event occurrences are permitted, what permissions are permitted, and so on.

## 6.2.4 Logic Programming and Datalog with Weak Negation

*Logic programming* languages are designed to operationalise various logics: logic programs encode logical theories. Here, we give an account of Datalog<sup>¬</sup> sufficient to understand the Datalog<sup>¬</sup> examples in Section 6.3.

Datalog, overviewed in [CGT89], is a simple logic programming language: each program is a set of *Horn clauses* called *rules*. Precisely, each rule has the form  $(c_1 \wedge c_2 \wedge \dots \wedge c_m \leftarrow a_1 \wedge a_2 \wedge \dots \wedge a_n)$ , where *consequents*  $c_1 \dots c_m$  and *antecedents*  $a_1 \dots a_n$  are *facts*, each of which is constructed from a *predicate* symbol  $p$  and a set of *constants* symbols and first-order variables over constants. Intuitively, each rule with variables encodes the set of variable-free rules with all variables consistently substituted with any conceivable combination of constants. The Datalog semantics gives each program a *model*, mapping *ground* facts (without variables) to Boolean values, which we interpret as distinguishing the facts that are *true* from *false*; each fact is true iff derivable by a rule in the program. Most of the literature and tools use the same concrete syntax: ( $\leftarrow$ ) and ( $\wedge$ ) are denoted ( $:-$ ) and ( $\cdot$ ), respectively, and only variable identifiers begin with uppercase letters. For example, `knows(amy,`

`Person` `:-` `knows(Person, amy)` formalises ‘each person known by Amy knows Amy’; note that `amy` is a constant, while `Person` is a variable. Implementations of Datalog all ultimately let users inspect the model.

Various dialects of Datalog have been studied in the literature, exploring the combination of various features. Datalog<sup>−</sup> [SZ94] is a useful generalisation: antecedents may be negated (with  $\neg$ , typically concretely denoted `not`), conditioning the truth of some facts on the falsity of others. This strictly improves expressiveness [KK20], because it affords *non-monotonic reasoning*: adding rules may *remove* truths [SA19]. For example, fact `sun` is true in program `sun :- not clouds`, but false after rule `clouds` is added; we say `sun` is *falsified*. Unfortunately, not all Datalog<sup>−</sup> programs have unique logical interpretations. Accordingly, different semantics exist (*e.g.*, stable model [GL88] and well-founded [VGRS91]) that attribute different models to these *unstratified* (defined in [Ros90]) programs. For example, what should be the value of `p` in `p :- not p`? Fortunately, we consider no such programs in this chapter.

Several tools can interpret (super-languages of) Datalog<sup>−</sup>. For example, the Clingo answer-set solver [GKK<sup>+</sup>11] can interpret each Datalog<sup>−</sup> example in this chapter.

### 6.3 The JustAct Framework for Multi-Agent Runtime Systems

This section defines the concepts and properties of the JustAct framework, along with explanations, motivations, and examples. We summarise the framework as follows:

*Agents make statements carrying policies that model the system.*

*Agents gossip and assemble statements into valid justifications.*

*Agents act only as permitted by their justifications.*

The essence of the framework is captured by the *framework ontology*, visualised in Figure 6.2: a collection of sets and functions capturing concepts and their relationships. We distinguish between *static* and *dynamic* concepts, and discuss each in a dedicated subsection. The framework is to be *instantiated* by a runtime system that defines the framework’s sets and functions such that the *framework requirements* are preserved; together, Sections 6.4 and 6.5 define such an instantiation. The most fundamental framework requirements are expressed in the figure; for example, each *action* has exactly one *justification*, and each *agreement* is a *message*. The remaining framework requirements are introduced throughout this section. The framework guarantees Properties 6.1 and 6.2, *i.e.*, agents always agree on action *permission* and *effects*, which are both precisely defined in Section 6.3.2.

Ultimately, we explain how agents can act autonomously while enforcing the *well-behavedness* (Definition 6.1) of themselves and their peers in practice.

**Property 6.1.** Agents always agree whether a given action is permitted.

**Property 6.2.** Agents always agree on the effects of a given action.

**Definition 6.1.** An agent is *well-behaved* while all their actions are permitted.

The rest of the section discusses the framework in general and incrementally builds an example framework instantiation using policies built from only the Datalog<sup>⊃</sup> rules in Table 6.1. We use a running example of well-behaved agents Amy and Bob. Ultimately, Bob enacts the deletion of Data1, justified by the policy  $\{r_d\}$ .

**Notation 6.1** (key notation underlying our definitions and properties).

- $\phi \triangleq \psi$  defines  $\phi$  as the expression  $\psi$ .
- $\phi = \psi$  proposes the equality of  $\phi$  and  $\psi$ , *i.e.*, this proposition may be false.
- $\psi \uplus \phi$  denotes the disjoint union of sets  $\phi$  and  $\psi$ . *I.e.*, if  $\phi \cap \psi = \emptyset$  then  $\uplus \triangleq \cup$  and otherwise  $\uplus$  is undefined. We use  $\uplus$  to express disjointness in our definitions.

### 6.3.1 Statics

The framework statics are fixed at runtime. They concern the fundamental concepts: data types and the syntax and semantics of the policy language. Agents reason about structure and relationship between given statics without having to communicate.

The **agents**<sup>1</sup> of the framework author **messages** occurring dynamically as statements, *i.e.*, the messages are fixed statically, and some subset of messages are *stated*

<sup>1</sup>Unless otherwise specified, let these agents coincide with agents of the distributed system.

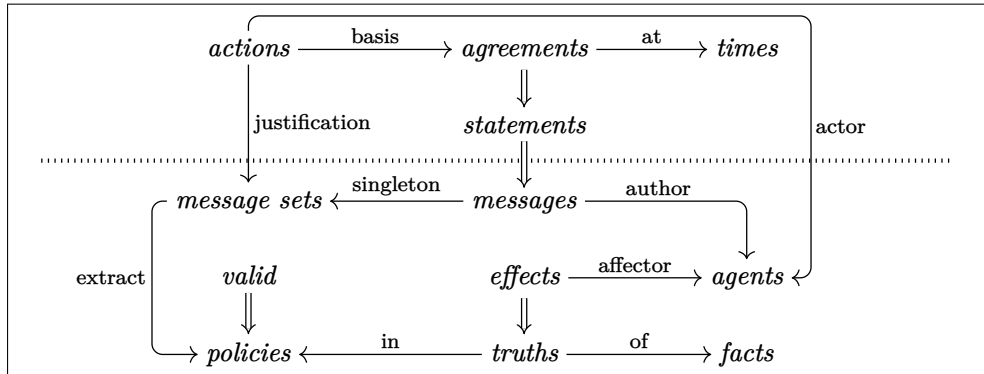


Figure 6.2: Graphical depiction of the *framework ontology*. Sets (italicized) are related by pure, total functions ( $\rightarrow$  and  $\Rightarrow$  arrows) from domain to co-domain. Each ( $\Rightarrow$ ) denotes an identity function, *e.g.*,  $statements \subseteq messages$ . Functions are identified by their co-domain (or by a label if given). The dotted line distinguishes sets and functions that are *dynamic* (above) and *static* (below). At runtime, new elements may be added to dynamics, but statics are fixed.

per runtime state. Statements communicate **policies** (see **extract** a few paragraphs down). We take for granted that ( $\in$ ) and *singleton* relate messages to *message sets* as expected. The **truths** define a relation over facts and policies, whose membership we test with  $true(f, p) \triangleq \exists t \in truths : of(t) = f \wedge in(t) = p$ . Alternatively, each policy prescribes which facts are true. In the examples throughout this section, we instantiate policies as the subsets of the (*policy*) *rules* shown in Table 6.1. Hence, the usual *set union* ( $\cup$ ) is a natural policy-composition operator, adding rules together. But note that the union of rules does not generally yield the union of truths. We use the semantics of Datalog<sup>-</sup>, as it is informally explained in Section 6.2.4, to define facts and truth:  $f$  is true in  $p$  iff  $f \in [p]$ , the *stable model* of policy  $p$ , e.g., when  $f = \mathbf{error}$  and  $p = \{r_m\}$ . Finally, **effects** are a special subset of truths that each identifies its **affector** agent. In Section 6.3.2, we explain how effects are the foundation for the interface between the runtime system and the outside world. In our example instantiation, we let (only) truths of `ctl-deletes(bob, data1)` be effects, affected by Bob.

The definition of **valid** fixes the distinction between useful and useless policies, as a function of their contents. We instantiate **valid** to characterise policies without errors; precisely, we let  $p \in valid \triangleq \neg true(\mathbf{error}, p)$ . Despite the simplicity of this definition, (in)validity emerges from complex rule interactions, which is key to policies capturing the complex relationships between domain concepts. As a simple example, no valid policy is a superset of  $\{r_m, r_n\}$ , as these rules suffice to ensure that **error** is tautological, i.e., true in any case. Intuitively,  $\{r_m, r_n\}$  captures the requirement that there must be a single consistent answer to the question: must Amy confirm? Section 6.6.2.2 demonstrates how rules conditioning (in)validity lay the groundwork for complex inter-agent power dynamics.

Most directly, **extract** relates messages to their policy contents. However, it also determines which policies agents are capable of expressing. For example, agent  $a$  simply cannot express  $p$  if *extract* maps no statements to  $p$ . Intuitively, *extract* can ensure that the extracted policy reflects its author(s). Hence, we understand the policy  $extract(m)$  as the subjective assertions of agent  $author(m)$ . For our running

Name	In natural language	As Datalog <sup>-</sup> rules
$r_m$	Amy <u>must</u> confirm.	<code>error :- not ctl-confirms(amy).</code>
$r_n$	Amy must <u>not</u> confirm.	<code>error :- ctl-confirms(amy).</code>
$r_c$	Amy <u>confirms</u> ...	<code>ctl-confirms(amy)</code>
	... if Amy trusts Bob.	<code>:- ctl-trusts(amy, bob).</code>
$r_t$	Amy <u>trusts</u> Bob.	<code>ctl-trusts(amy, bob).</code>
$r_d$	Bob <u>deletes</u> Data1.	<code>ctl-deletes(bob, data1).</code>

Table 6.1: Example (*policy*) *rules* expressed in Datalog<sup>-</sup> and natural language. Each rule’s name is suggested by the underlined letter. A policy is any set of these rules.

example with  $\text{Datalog}^-$ , we instantiate  $\text{extract}$  as a pure function of the message author and the message  $\text{payload}$ , which is an arbitrary policy, chosen by the author. Intuitively,  $\text{extract}$  invalidates payloads on the condition that they include any undesirable author-rule pairs. Precisely, we let  $\text{extract}(\{m\}) \triangleq \text{payload}(m) \cup \{\text{error} \mid \forall (r, a) \in \text{owns} : (r \in m \wedge \neg \text{author}(m, a))\}$ , where  $\text{owns}$  relates rule  $r$  to agent  $a$  if  $r$  has an antecedent whose predicate is prefixed by  $\text{ct1-}$  and whose first parameter is constant  $a$ . For example, only Bob owns rule  $\text{ct1-deletes}(\text{bob}, \text{data1})$ , while rule  $\text{error} :- \text{confirms}(\text{amy})$  has no owner because fact  $\text{error}$  has neither prefix  $\text{ctr1-}$  nor a constant first parameter. Throughout this chapter, we always close the definition of  $\text{extract}$  such that it distributes over messages. Precisely, we let  $\text{extract}(\emptyset) \triangleq \emptyset$ ,  $\text{extract}(\{m\} \uplus M) \triangleq \text{extract}(\{m\}) \cup \text{extract}(M)$ . This definition confers a property that simplifies reasoning:  $\text{extract}$  commutes and associates over message set union. As agents agree on the definition of  $\text{extract}$ , its design affords the opportunity to agree on the fundamental limitations on which subjective assertions are available to each agent. For example, for any message set  $M$  where  $\text{true}(\text{ct1-deletes}(\text{bob}, \text{data1}), \text{extract}(M))$ , either  $M$  includes messages authored by Bob, or  $\text{extract}(M)$  is invalid. Intuitively, Amy alone cannot assert that Bob deletes Data1.

### 6.3.2 Dynamics

Elements of framework dynamics are fixed in each system configuration at runtime. They instantiate the statics and must be stored and communicated by agents. Figure 6.3 illustrates the agents' partial views on some dynamic system configuration.

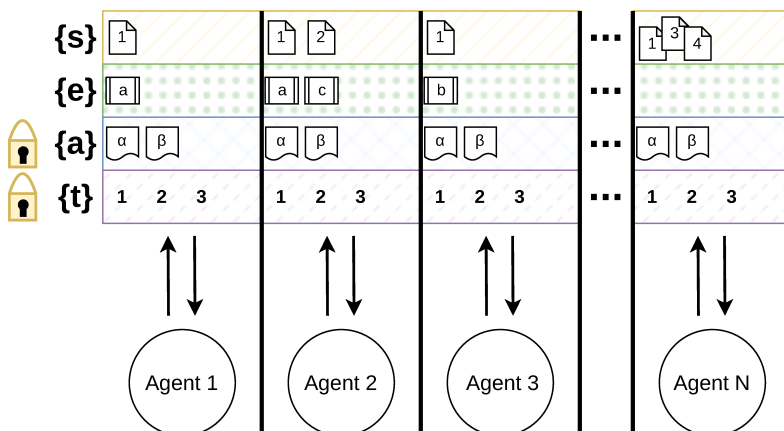


Figure 6.3: Graphical depiction of a dynamic configuration, where each agent has a partial view of all (s)tated messages, (e)nacted actions, (a)greements, and current (t)imes. Agents have the (same) global view of the agreements and current times.

At runtime, a subset of messages are **statements** because they have been stated, and then potentially shared with other agents. Intuitively, statements have few restrictions, but therefore, they have little impact on their own. They are meaningful because of their relationship with actions. In practice, we expect each statement to be *autonomously* created by its author, *i.e.*, as decided by the author alone. In theory, the set of statements grows forever. In practice, agents and networks have finite storage capacities, so we let agents forget or discard statements. For our purposes, it suffices if the existence of a given statement is semi-decidable: an agent can decide that statement  $m$  exists by observing it, and can prove its existence to other agents by showing them  $m$ . In practice, the integrity and provenance of statements is preserved by authors cryptographically signing their statements, and agents ignoring statements without their authors' signatures. Agents share statements with their peers (*e.g.*, via gossip), for their own reasons, and at their own pace. Section 6.6 demonstrates a case where an agent intentionally withholds a statement from other agents.

**Agreements** attribute special meaning to selected statements at selected **times**. We require that membership be decidable, *i.e.*, Property 6.3. Intuitively, this lays the groundwork for agents agreeing which actions are permitted. In practice, agents explicitly synchronise the growing set of agreements. In real systems, synchronisation may be infrequent and agreements may be expressed sparsely; *e.g.*, one synchronisation creates agreements for each time from 100 to 500. But for clarity, we represent and synchronise individual agreements explicitly. In our running example, Amy and Bob agree on a statement with policy  $\{r_m\}$  that applies at time 1. In general, it is crucial for the runtime system to preserve Property 6.3: agents must always agree which subset of times are *current*. We call agreements current if their time is current. Thus, agents can add and remove current agreements over time, but they maintain agreement on the growing set of agreements per fixed time. We expect agents to rely on this to robustly reason about the agreements at prior and hypothetical future times. For example, even after the current time is changed to 2, Amy and Bob agree on the agreements at time 1, and that those are not current. As we expect it to be practical in general, in our examples, we let agents maintain a single current time, *i.e.*, the current time is a variable that agents synchronously overwrite.

**Property 6.3.** Agents always agree on the current times and the (current) agreements.

At runtime, there is a growing set of (taken) **actions**. Each action has a defined *actor* agent, justification<sup>2</sup> (a set of messages), and *basis* agreement. We say the action is taken at the time of its basis. Like statements, actions are created autonomously (by their actors) and then shared between agents by unspecified gossip. Unlike

<sup>2</sup>In the first article presenting JustAct [EMvB24], one message in the justification is *enacted*, determining the action effects. Here, we instead unify the justification and enacted statements. These can be distinguished again at the user level, via policy, if necessary.

statements, actions are restricted such that they can be effectful and meaningful. Precisely, *well-behaved* agents take only *permitted* actions. Well-behavedness has only extrinsic value, because agents trust and expect their peers to be well-behaved and may suffer external consequences otherwise. For example, by agreeing on the statement with payload  $\{r_m\}$ , Amy and Bob agree that actions are not permitted without Amy’s confirmation. Amy can choose to trust Bob to remain well-behaved by accessing `Data1`, assuming it will not be deleted unexpectedly. All observers agree that Bob is not permitted to enact the deletion of `Data1` without Amy’s confirmation.

**Definition 6.2.**  $well\text{-}behaved(\alpha : agents) \triangleq$   
 $\forall a : actions, (actor(a) = \alpha) \rightarrow permitted(a).$

**Definition 6.3.**  $permitted(a : actions) \triangleq$

$justification(a) \subseteq statements$	(stated justification)
$\wedge agreed(basis(a)) \in justification(a)$	(based justification)
$\wedge extract(justification(a)) \in valid$	(valid justification)
$\wedge at(basis(a)) \in current.$	(current action)

Here we break down and explain Definition 6.3: whether a given action is permitted, in the context of a fixed (dynamic) system configuration. Firstly, justifications must consist entirely of **stated** messages. This ensures that justifications reflect the existing assertions of their respective authors, *i.e.*, and not arbitrary policies in conceivable messages. Hence, agents restrict actions by withholding their statements. For example, Amy cannot justify actions in terms of Bob’s hypothetically-stated messages; Bob must state them first! Secondly, justifications must be **based**, *i.e.*, include their basis agreement. Thus, the meaning of a justification is regulated by its basis. For example, `ct1-trusts(amy, bob)` is necessarily true in all justifications based on  $\{r_t\}$ . Thirdly, (the policy extracted from) each justification must be **valid**. These justifications have internally-consistent meanings. For example, recall that only Bob owns  $r_d$ . Hence, Amy can make statements whose payloads include  $r_d$ , but those statements can be safely ignored, as they are useless in justifying permitted actions. In general, definitions of validity are useful if they enable meaningful reasoning about policies in terms of their parts. For example, for any conceivable policy  $p$ , the composite policy  $\{r_m, r_n\} \cup p$  is invalid, because  $r_m$  and  $r_n$  place contradictory conditions on validity: Amy cannot both confirm and not confirm! Finally, actions must be **current**: based on a current time. This is the only component of permission that may be falsified in the future, *i.e.*, all the other components that hold at present necessarily hold in the future. In general, we expect agents to reason about historical and hypothetical system configurations; they come to reliable conclusions. For example, Amy and

Bob agree to change the current time from 1 to 2. Actions permitted at time 1 are unaffected, but they are no longer permitted at the current time. Instead, new actions are permitted, based on the agreement  $\{\}$  at time 2. Section 6.6.2.2 lays out a realistically complex agreement and discusses its underlying intuition by reasoning about the actions it permits. Thereafter, Section 6.6.3.5 demonstrates changing this agreement by adding a changed agreement and changing the current time.

Ultimately, Definition 6.4 defines the (enacted) effects of a given action. Like actions, these only grow at runtime. However, where actions are user-defined during instantiation (hence, actions are included in Figure 6.2), the definition of enacted effects is fixed by Definition 6.4, derivable from the actions themselves; hence, *e.g.*, agents can communicate enacted effects implicitly by explicitly communicating actions. Enacted effects are intended to trigger events external to the system; we say these effects are *realised*. For example, Bob realises the enacted effect `ct1-deletes(bob, data1)` by deleting the asset data identified by `data1` from an external database. Note that the framework distinguishes actors (which collect justifications, take actions, and affect well-behavedness) from *affectors* (which realise effects). This admits cases where these roles are decoupled. However, in this chapter, instantiations and agreements are designed such that, in all permitted actions, actors and affectors coincide.

**Definition 6.4.** *enacted-effect-of*( $e : \text{effects}, a : \text{actions}$ )  $\triangleq$   
 $\text{in}(e) = \text{extract}(\text{justification}(a)).$

Ultimately, the utility of the framework is how it systematises the cooperation between agents using policies to define, constrain, and enact the permitted actions and their effects. This is achieved despite actions and statements being autonomously created and gossiped. Hence, agents can work towards unrelated actions concurrently. Nevertheless, by fixing the semantics of the policies, and by synchronising the agents' views on the current agreements, the agents can rely on Properties 6.1 and 6.2: the agents maintain agreement on the permission and effects of every conceivable action. Well-behaved agents can work toward realising particular, desirable effects by reasoning backwards from the effect to the necessary actions, and then further to the necessary times, agreements, and statements. Our running example is concluded as Bob takes a final action at time 2, based on the trivial agreement, and justified by policy  $\{r_d\}$ . Amy, Bob, and future auditors all come to the same conclusions: this action is permitted and has the effect `dt1-deletes(bob, data1)`.

In general, there are many approaches to enforcing the well-behavedness of agents. In fact, we expect a mix of approaches to be used in practice. We expect benevolent agents to systematically enforce their own well-behavedness via *ex-ante* enforcement; *i.e.*, these agents check for permission before acting. Cautious agents can await proof of permission from their peers before accepting their actions. Agents with

limited resources can trust their peers by assuming that their actions are permitted, instead relying on auditors to punish peers for bad behaviour after the fact. We consider it sensible for agents to be accountable for their own actions; 1. actors must always be ready to prove that their actions are justified, and 2. actors cannot hide their actions, *e.g.*, to feign well-behavedness. Additional safety can be achieved via redundancy; for example, agents periodically audit suspicious actions, and agents assist auditors by reporting which actions they observed. Section 6.4 discusses the choice and implementation of enforcement mechanisms, and then Section 6.6 demonstrates particular agents enforcing well-behavedness in particular scenarios.

## 6.4 Implementation: Generic Data Exchange Runtime System

In this section, we design and implement a prototype multi-agent data exchange runtime system for use in our case study. It is architected around two complementary parts, as visualised in Figure 6.4:

1. **The Data Plane:** Agents are motivated to create and share (*data*) *assets* with their peers. This is done by agents reading and writing assets in a shared store.
2. **The Control Plane:** Agents create and share meta-data such as policies. This meta-data ultimately regulates agents' access to assets in the data plane.

We instantiate the JustAct framework as the control plane. The interface between the two planes corresponds to the interface between the JustAct framework and its runtime environment: JustAct reading and writing *effects* in the control plane are *realised* as agents reading and writing data assets in the data plane.

This section leaves the JustAct statics relating to the *policy language* uninstantiated; hence, in isolation, this section contributes a runtime system that remains generic to the policy language. Section 6.5, to follow, completes the instantiation by fixing Slick as the particular policy language. It is this complete instantiation which we use for the case study in Section 6.6.

### 6.4.1 Design Considerations

Data exchange systems are subject to many design considerations before they can be implemented in detail. Here, we explore particular design considerations, and explain the decisions reflected in our prototype implementations.

Firstly, **how is the (meta) data managed?** Our prototype system centralises (meta) data stores within shared memory. This simplifies the data exchange implementation itself, such that we can focus on the main concern of our contribution: (regulating) the interactions between the agents. For example, it trivialises the

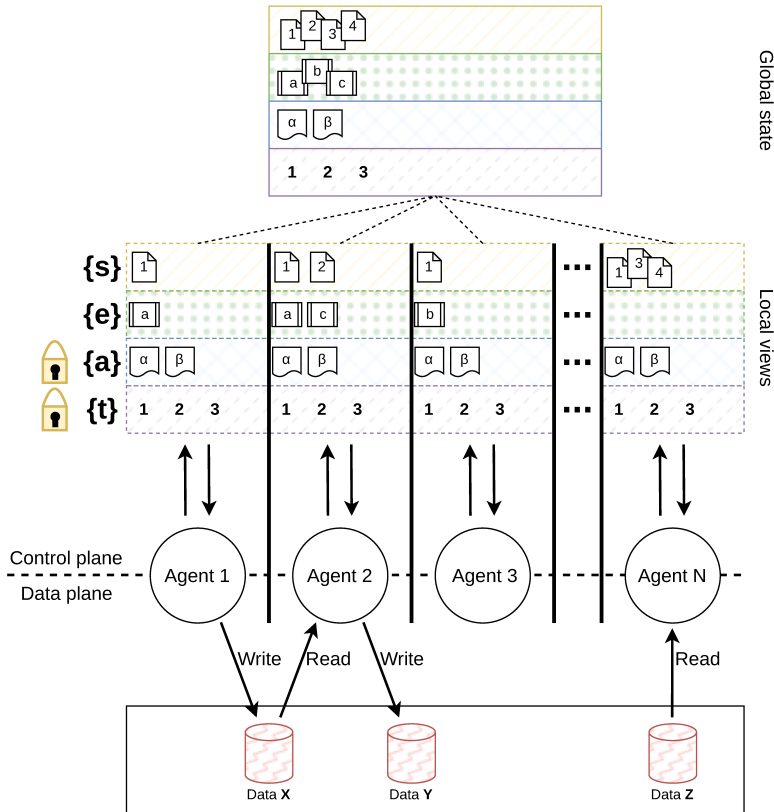


Figure 6.4: Graphical depiction of a dynamic system configuration implemented by our prototype. The control plane (above) inherits the architecture of the JustAct framework dynamics shown in Figure 6.3: agents have partial views on control-level meta data. Agent views are implemented as masks over a central meta data store. In the data plane (below), agents read and write data in another shared store.

(de)serialisation of inter-agent communications, and the collection of diagnostic information during experiments, *e.g.*, we can easily inspect entire system snapshots to compare the views of different agents. Realistic data exchange system implementations would instead require physically distributed stores of (meta) data, which requires further design decisions (*e.g.*, how are agreements synchronised?) which we leave out of scope. In anticipation of future exploration of these concerns, our implementation preserves the distributed system abstraction. For example, an agent cannot observe a statement it has not created or received from a peer.

Secondly, **how is well-behavedness defined?** Our implementation adopts the view of the JustAct framework: well-behaved agents take only permitted actions, where permission is regulated by agents' policies. We postpone the details of the policies until Section 6.5, where we characterise and demonstrate the Slick language.

Finally, **how is well-behavedness enforced?** While the JustAct framework defines permission and well-behavedness via policies, the implementation defines how agents enforce well-behavedness. In general, many approaches are possible. When misbehaviour is *prevented* (ex-ante enforcement), agents can reason in absolutes about (im)possibility via (im)permissibility, *e.g.*, affording their assessment of the risks of harm incurred by misbehaviour. When misbehaviour is *punished* (ex-post enforcement), it has consequences for agents' behaviour in the future. We leave these considerations out of scope, and implement an enforcement mechanism that suffices to support our experiments and emphasise our unified view of permission: agents can behave arbitrarily, but every statement, action, and effect is logged. This suffices to enable perfect external audits of permission and misbehaviour after the fact. Our artefact includes a supplementary tool for performing and visualising these diagnoses. An example of the visualisation is shown in Figure 6.5. Enforcement in the data- and control-planes are discussed in Sections 6.4.2 and 6.4.3, respectively.

## 6.4.2 The Data Plane

Agents are ultimately motivated to interact with the data plane, where their data is stored and shared. By our shared-memory implementation, the sharing of data between agents is trivial; each agent has a view on the same data store.

Agents' *read* or *write* (together: *access*) data in the shared data store. The runtime system regulates these events by requiring that each realises an *enacted effect* in the control plane. We define (*enacted*) *effects* precisely in Section 6.5.3.1 atop the policy language. But here, it suffices to say that agents must contextualise each of their access events by identifying the corresponding action (explicitly), data (explicitly), and affector (implicitly: themselves). Agents are able to perform any access events, but the runtime system logs this behaviour along with its context.

The runtime system realises the intended correspondence between access events and enacted effects. After the fact, auditors are able to determine precisely which access events *realise* enacted effects. Section 6.4.3 follows by detailing the related concerns of the control plane, including the creation and permission of actions.

## 6.4.3 The Control Plane

The control plane implements the majority of the concepts of the JustAct framework as straightforwardly as possible. Here, we discuss their implementation *in abstracto*, focusing on the essence of the implementation, before Section 6.4.3.5 presents the implementation in more detail, as Rust code snippets.

**JustAct Prototype Event Inspector - v0.1.0**

---

~Event

- 1) [JUSTACT] Agent st-antoniuss stated message "st-antoniuss 1"
- 2) [JUSTACT] Agent surf stated message "surf 1"
- 3) [JUSTACT] Agent consortium stated message "consortium 1"
- 4) [JUSTACT] Published agreement "consortium 1"
- 5) [JUSTACT] Advanced to time 1
- 6) [JUSTACT] Agent amy stated message "amy 1"
- 7) [JUSTACT] Agent dan stated message "dan 1"
- 8) [JUSTACT] Agent st-antoniuss enacted action "st-antoniuss a" ✓
- 9) [DATAPLN] Agent st-antoniuss wrote to variable "(st-antoniuss patients-2024)"
- 10) [JUSTACT] Agent surf enacted action "surf a" ✓
- 11) [DATAPLN] Agent surf wrote to variable "(surf utils) entry-count" ✓
- 12) [JUSTACT] Agent surf stated message "surf 2"
- 13) [JUSTACT] Agent st-antoniuss stated message "st-antoniuss 2"
- 14) [JUSTACT] Agent st-antoniuss enacted action "st-antoniuss b" ✓
- 15) [DATAPLN] Agent st-antoniuss read variable "(surf utils) entry-count" ✓
- 16) [DATAPLN] Agent st-antoniuss read variable "(st-antoniuss patients-2024) pat"
- 17) [DATAPLN] Agent st-antoniuss wrote to variable "(amy count-patients) num-pa"
- 18) [JUSTACT] Agent amy stated message "amy 2"
- 19) [JUSTACT] Agent st-antoniuss stated message "st-antoniuss 3"
- 20) [JUSTACT] Agent amy enacted action "amy a" ✓
- 21) [DATAPLN] Agent amy read variable "(amy count-patients) num-patients" ✓

~Event 8

Enacted by: **st-antoniuss**  
 Enacted to: **<everyone>**

Action identifier: **st-antoniuss a**  
 Action actor : **st-antoniuss**  
 Action taken at : **1**

Basis : **consortium 1**  
 Justification : **consortium 1 and st-antoniuss 1**

Permission : **OK**  
 Effects :

- (**st-antoniuss writes ((st-antoniuss patients-2024) patients)**)

~Justification truths

```

(((st-antoniuss patients-2024) executed) within (st-antoniuss 1))
(((st-antoniuss patients-2024) has output patients) within (st-antoniuss 1))
(((st-antoniuss patients-2024) involves st-antoniuss) within (consortium 1))
(((st-antoniuss patients-2024) ready) within (st-antoniuss 1))
((authorise (st-antoniuss patients-2024) in (st-antoniuss 1) by st-antoniuss) w
((st-antoniuss controls ((st-antoniuss patients-2024) patients)) within (st-an
((st-antoniuss drives (st-antoniuss patients-2024)) within (consortium 1))
((st-antoniuss patients-2024) executed)
((st-antoniuss patients-2024) has output patients)
((st-antoniuss patients-2024) involves st-antoniuss)
((st-antoniuss patients-2024) ready)
((st-antoniuss says ((st-antoniuss patients-2024) executed) within (consortiu
((st-antoniuss says ((st-antoniuss patients-2024) has output patients)) within
((st-antoniuss says ((st-antoniuss patients-2024) ready)) within (consortium 1
((st-antoniuss says (authorise (st-antoniuss patients-2024) in (st-antoniuss 1)

```

Press Q to quit

Press Esc to close trace

Press Shift+Tab to switch to list

Figure 6.5: Screenshot from the interactive trace inspector. On the left, a list of all system events (both JustAct/control plane and data plane) is shown, where the right shows a more detailed view of a particular enactment. It includes information on actorship, which statements are included in the justification and whether the policy extracted is valid and what are its effects.

### 6.4.3.1 Controlled Gossip of Stated Messages

The statics relating to messages are implemented as data types and (field) projection functions. Each message is a product of a *message identifier* and a *payload*: an arbitrary policy. Each message identifier is a pair of its author and a discriminator: arbitrary facts. In the definition of *author'* is the first of many times we use what we call the ‘prefix trick’ to ensure there exists a functional projection of type  $X \rightarrow Y$ ; we let  $X \triangleq Y \times Z$  for some discriminator  $Z$ , and project  $X$  to its first element.

**Definition 6.5** (Message Identifiers).  $messageIds \triangleq facts \times facts$ .

**Definition 6.6.**  $messages \triangleq messageIds \times policies$ .

**Definition 6.7.**  $payload(\langle i, p \rangle : messages) \triangleq p$ .

**Definition 6.8.**  $author(\langle i, p \rangle : messages) \triangleq author'(i)$   
 where  $author'(\langle f, f' \rangle : messageIds) \triangleq f$ .

The dynamic creation and communication of statements is implemented as a store of *sent* messages, which are exchanged from one peer to another. In this context, we disregard the recipients of the sent messages; *e.g.*, it relaxes the requirement on the network to deliver sent messages. While they are related, we distinguish message *senders* from *authors*. Only the latter is evident in the message itself, and then the same message can be sent repeatedly, *e.g.*, to different recipients at different moments in time. In real distributed systems, this decentralised *gossip* is a flexible and scalable means of disseminating non-critical information.

**Definition 6.9** (Sent Messages).  $sent \triangleq agents \times messages$ .

**Definition 6.10.**  $statements \triangleq \{m \mid \forall \langle \alpha, m \rangle \in sent\}$ .

The runtime controls the gossip of statements to preserve their intended interpretations. Property 6.4 ensures that all sent messages are (at least) sent by their authors. Hence, gossiped statements necessarily originate at their authors, and thus, express the authors’ subjective assertions. Property 6.5 identifies payloads via message identifiers. This lets agents define policies that refer to (other) policies. For example, near the end of Section 6.6.2, this is formalised in Part 5 of the initial agreement statement (Statement `consortium 1`): agents can *authorise* data-processing tasks, but these authorisations are only applicable with specified policies.

It is simple for the runtime system to systematically preserve Properties 6.4 and 6.5, because it manages the view of each agent; the system simply checks which messages the sender has previously received (for Property 6.4) and sent (for Property 6.5).

**Property 6.4** (Message Provenance).  $\forall \langle \alpha, m \rangle \in sent, \langle author(m), m \rangle \in sent$ .

**Property 6.5** (Identifiable Payloads).  $\forall \langle i, p \rangle, \langle i', p' \rangle \in statements, i = i' \rightarrow p = p'$ .

### 6.4.3.2 Controlled Gossip of Taken Actions

The JustAct concept of **action** is implemented similarly to that of statements. The static component of actions are a composite data type, the composition of their actors, justifications, and basis agreements. The dynamic component concerns the gossip of (taken) actions. As with statements, agents' requests to act are regulated to preserve provenance; agents can gossip about other agents' actions, but agents cannot *create* new actions on another actor's behalf.

Unlike statements, the creation of new actions is regulated. The runtime system enforces that only *permitted* actions are taken: after the fact, auditors can evaluate the meta data in the system trace, and check which actions were permitted when they were created. Figure 6.5 demonstrates such an audit, confirming that the action under inspection indeed satisfies all four conditions of permission (see Definition 6.3).

### 6.4.3.3 Synchronous Time and Agreement Updates

To preserve Property 6.3, the runtime system specialises the handling of changes to *asynchronous* and *synchronous* control meta data. Two kinds of changes to synchronous meta data are possible: 1. the current time is changed to the given time, and 2. the given agreement is added. These changes have wide-reaching consequences for which actions are permissible. Hence, they are carefully regulated, and are intended to be used sparingly. For the sake of the experiments, we simply vest the power to make synchronous changes in a particular agent. We call this agent *consortium*, to reflect its role in expressing the consensus of all agents.

### 6.4.3.4 Scripted Agent Behaviour

Each agent interfaces with the control plane via a dedicated, bidirectional communication channel: agents send requests to state messages and take actions, and agents receive asynchronous gossip from their peers, and can read the synchronised control data, for example, to check the current time.

For the sake of the reproducibility of our experiments, we implement a simple agent-scripting layer: the behaviour of each agent is fixed by an event-handler, defining how the agent reacts to environmental events by stating messages and taking actions. For example, an agent can be scripted to react to the statement of a given message, a given fact becoming true in any known statement, a given action becoming permitted, and so on. Thus, our accompanying artefact includes a suite of unit tests which lets users reproduce and inspect each of our case study scenarios in Section 6.6.3. Precisely, our agent scripting language is embedded in Rust, atop the runtime implementation itself. Listing 6.1 shows an example script (which specifies Amy's behaviour to be

Listing 6.1: An example script of an agent’s behaviour, embedded in Rust, but simplified for readability. The script demonstrates stateful reasoning, reacting to an agreement in the control plane by taking action. This script defines the behaviour of agent Amy in Section 6.6.3.1.

```

self.handler
// Wait for the fact
// `(surf utils) ready.`
// publish `amy 1`.
.on_truth(
  slick::parse::ground_atom(
    "(surf utils) ready"
  ),
  |view| view.stated.add(
    Recipient::All,
    create_message(1, "amy", r#"
(amy count-patients)
  has input ((surf utils) entry-count).
(amy count-patients) has input
  ((st-antoniuss patients-2024) patients).
(amy count-patients)
  has output num-patients.
(amy count-patients) ready."#)
  )
)
// Wait for the action `st-antoniuss b`,
// publish `amy 2`.
.on_enacted(
  ("st-antoniuss", 'b'),
  |view, _| view.stated.add(
    Recipient::All,
    create_message(2, "amy", r#"
(amy end) has input
  ((amy count-patients) num-patients).
(amy end) ready.
(amy end) executed."#)
  )
)
// Wait for agreement and selected
// statements, publish `amy a`
.on_agreed_and_stated(
  ("consortium", 1),
  [
    ("amy", 1), ("amy", 2),
    ("st-antoniuss", 1),
    ("st-antoniuss", 2),
    ("st-antoniuss", 3), ("surf", 1),
  ],
  |view, agree, just| {
    view.enacted.add(
      Recipient::All,
      create_action(
        'a', "amy",
        agree, just
      )
    );
    self.store.read(
      (
        "amy", "count-patients",
        "num-patients"
      ),
      ("amy", 'a')
    );
    Ok(())
  })?
.finish()

```

discussed in Section 6.6.3.1) which embeds policies (expressed in the Slick language to be discussed in Section 6.5). Amy’s behaviour is expressed as an event handler constructed from a sequence of condition-guarded reactions to the environment. For example, Amy reacts to the observation of a statement in which `(surf utils) ready` is true by making Statement `amy 1`, *i.e.*, a message with identifier `amy 1` and author `amy`.

We expect realistic systems to automate the behaviour of some agents in a similar fashion. The details of the automation are subject to a large design space. For example, scripts that await given messages are less costly to execute, while scripts that evaluate statements are more flexible and expressive. The most intelligent agents must formulate and solve complex search problems over statements and actions.

### 6.4.3.5 Overview of the Implementation

The prototype runtime system is implemented in Rust, and the implementation of its control plane is structured to reflect the structure of the chapter: the Rust

Listing 6.2: Simplified Rust source code showing a part of the control plane's generic ontology.

```

trait Message {
    type Id;
    type Payload;
    fn id(&self) -> &Self::Id;
    fn payload(&self) -> &Self::Payload;
}
struct MessageSet<M: Message> {
    msgs: HashMap<M::Id, M>
}
trait Timestamp: Eq + Ord {}
trait Action {
    type Message: Message;
    type Timestamp: Timestamp;
    fn basis(&self) -> Agreement<
        Self::Message,
        Self::Timestamp
    >;
    fn justification(&self)
        -> MessageSet<Self::Message>;
}
struct Agreement<M: Message, T: Timestamp>
{ message: M, timestamp: T }

trait Set<E> {
    fn contains(&self, elem: &E) -> bool;
    fn iter(&self)
        -> impl Iterator<Item = &E>;
}
// Encodes centralized mutability
trait SetSync<E>: Set<E> {
    fn add(&mut self, elem: E);
}
enum Selector { All, Agent(String) }
// Encodes distributed mutability
trait SetAsync<E>: Set<E> {
    fn add(
        &mut self,
        target: Selector,
        elem: E
    );
}
trait Times: Set<Self::Timestamp> {
    type Subset: Set<Self::Timestamp>;
    type Timestamp: Timestamp;
    fn current(&self) -> Self::Subset;
}
trait TimesSync: SetSync<Self::Timestamp>{
    fn add_current(
        &mut self,
        timestamp: Self::Timestamp
    );
}

trait Agent {
    fn id(&self) -> &str;
    fn poll<ME, AC, TS, ST, EN, AG, TI>
        (&mut self, view: (ST, EN, AG, TI))
    where
        ME: Message,
        AC: Action<Message = ME,
            Timestamp = TS>,
        TS: Timestamp,
        ST: SetAsync<ME>,
        EN: SetAsync<AC>,
        AG: Set<Agreement<ME, TS>>,
        TI: Times<Timestamp = TS>;
}
trait Synchronizer {
    fn poll<ME, AC, TS, ST, EN, AG, TI>
        (&mut self, view: (ST, EN, AG, TI))
        -> ControlFlow;
    where
        AG: SetSync<Agreement<ME, TS>>,
        TI: TimesSync<Timestamp = TS>,
        /* Rest is same as for agents */;
}

```

embedding of the JustAct framework (in general) is loosely coupled to its particular implementation in our prototype data exchange runtime system (in particular).

First, Listing 6.2 shows a simplified version of the *ontology* library, encoding the JustAct framework as Rust traits. It specifies the capabilities of information exchanged between agents (message(set)s, actions, timestamps and agreements) and of sets storing that information (stated, enacted, times and agreed). An abstraction of agents and entities capable of synchronization is given to be able to interchange them between specific runtimes.

Second, Listing 6.3 implements the traits comprising the ontology, *i.e.*, instantiating the corresponding concepts of the JustAct framework. For example, for simplicity (and efficiency), the following implementation passes control information

within shared memory, rather than (de)serialising and passing it over the network. The distributed system is simulated by each agent having its own *view* of the control plane as a subset of the statements and actions. Both the global and local views on statements and actions are implemented as *grow-only sets*, which are *contention-free replicated data types (CRDT)*, which are suited for adaptation to implementations in which agents are physically distributed [ASB15]. Note that the implementations of the traits are orthogonal to the implementation of agent behaviour: that is still done over the interfaces only, showing that agents can reason at the level of the framework instead of the underlying implementation details.

Finally, Listing 6.4 brings the implemented structs together by orchestrating the agents and their views on the system. The implementation of the runtime system becomes sufficiently complete for compilation once it is (statically) instantiated with a particular policy language.

## 6.5 The Slick Policy Language & Interpreter

We use *policy language* to refer to a set of definitions of JustAct concepts concerning policies. The definition of *policy* captures the language syntax. The definitions of *truth* (via *fact*, *in*, and *of*) and *valid* capture orthogonal semantic notions of policies. So Sections 6.4 and 6.5 together complete an instantiation of the JustAct framework.

### 6.5.1 Design Considerations

The framework imposes several essential requirements on the policy language. The most essential requirement is expressed in Figure 6.2 itself: truth must be a static relation over facts and policies. In practice, this means that the truth of a given fact in a given policy must be computable via a deterministic and terminating procedure. This requirement underpins the purpose of the same policies to be consistently understood by different agents, or at different times. For example, (complete) Prolog is an unsuitable policy language, because its inference procedure is non-terminating. Even if agents agree on a Prolog program, their conclusions are not consistently reproduced. Consider a case where Amy justifies the processing of Bob's medical data with a Prolog policy which Amy successfully validated. Three hours later, an auditor checks the justification, but their evaluation of the Prolog policy times out. The auditor cannot distinguish between an invalid policy and a policy that Amy intentionally crafted to time out. Should Amy be punished for acting? In such an environment, would Amy risk acting in the first place?

Listing 6.3: Simplified Rust source code showing a part of the control plane's specific instantiation.

```

struct MessageData {
    id: (String, u32),
    payload: Policy,
}
impl justact::Message for MessageData {
    type Id = (String, u32);
    type Payload = Policy;
    fn id(&self) -> &Self::Id { &self.id }
    fn payload(&self) -> &Self::Payload {
        &self.payload
    }
}

impl justact::Timestamp for u64 {}

struct ActionData {
    basis: Agreement<MessageData, u64>,
    justification: MessageSet<MessageData>,
}
impl Action for ActionData {
    /* akin to MessageData */
}

struct AsyncSet<E> {
    data: HashMap<String, HashSet<E>>
}
impl<E> AsyncSet<E> {
    // Returns the view of agent `id`
    fn scope(&mut self, id: String)
        -> AsyncSetView<E> {
        AsyncSetView { set: self, id }
    }
}

struct AsyncSetView<E> {
    set: &mut AsyncSet<E>,
    agent: String,
}
impl<E> Set<E> for AsyncSetView<E> {
    fn contains(&self, elem: &E) -> bool {
        self.set.data.get(&self.agent)
            .contains(elem)
    }
}

fn iter(&self)
    -> impl Iterator<Item = &E> {
    self.set.data.get(&self.agent)
        .iter()
} }
impl<E> SetAsync<E> for AsyncSetView<E> {
    fn add(
        &mut self,
        selector: Selector,
        elem: E
    ) { match selector {
        Selector::All => {
            for view in &mut self.set.data {
                view.insert(elem.clone());
            }
        },
        Selector::Agent(agent) => {
            self.set.data.get_mut(&agent)
                .insert(elem)
        }
    } } }

struct TimesSet {
    all: HashSet<u64>,
    current: Option<u64>,
}
impl Set<u64> for TimesSet {
    /* maps to `self.all` */
}
impl SetSync<u64> for TimesSet {
    /* maps to `self.all` */
}
impl Times for TimesSet {
    type Subset = Option<u64>;
    type Timestamp = u64;
    fn current(&self) -> Self::Subset {
        self.current
    }
}
impl TimesSync for TimesSet {
    fn add_current(
        &mut self,
        timestamp: Self::Timestamp
    ) {
        self.current = Some(timestamp);
    }
}

```

Listing 6.4: Simplified Rust source code showing the ‘game loop’ of the system, orchestrating the agents and managing their access to state.

```
fn game_loop(
  agents: Vec<dyn Agent>,
  sync: Box<dyn Synchronizer>
) {
  let st = AsyncSet::<Message>::new();
  let en = AsyncSet::<Action>::new();
  let ti = TimesSet::new();
  let ag =
    HashSet::<Agreement<Message, u64>>
      ::new();

  loop {
    for agent in &mut agents {
```

```
      agent.poll((
        &mut st.scope(agent.id()),
        &mut en.scope(agent.id()),
        &ag,
        &ti,
      ));
    }

    match sync.poll((
      &mut st.scope("environment"),
      &mut en.scope("environment"),
      &mut ag,
      &mut ti,
    )) {
      ControlFlow::Continue => continue,
      ControlFlow::Break => break,
    } } }
```

## 6.5.2 The Slick Language

The Slick language was first defined in Section 4.5.3, following a study of various languages and their variants. To minimise the coupling of our (articles underlying) the thesis chapters, here we sufficiently (re)define Slick in terms of Datalog<sup>⊥</sup>.

Slick is a (policy) specification language that was designed with cooperative specification in mind. Slick and Datalog<sup>⊥</sup> have many similarities: each program is a sequence of rules (but whose ordering is irrelevant to the semantics) whose antecedents are possibly negated. For each policy, the semantics prescribes Boolean truth to a finite subset of the infinite syntactic category of ground (*i.e.*, variable-free) facts. We complete the description of Slick by enumerating its differences from Datalog<sup>⊥</sup>; observe that each difference is either superficial or based on well-established research:

1. Slick departs from the traditional **concrete syntax** of Datalog<sup>⊥</sup> to (in our opinion) aid in the readability of long rules. We conjoin atoms with **and** and arguments of atoms with white space (both instead of `(,)`) and we separate consequents from antecedents by **if** instead of `(:-)`. The use of **not** is unchanged, as it already suits the new style.
2. **Inference** uses the operational form [Prz90] of the *well-founded semantics for general logic programming with negation* [VGRS91]; a value in `{true, false, unknown}` is deterministically assigned to every conceivable policy. Notably, the semantics robustly preserves meaning even in cases that many alternative inference semantics (*e.g.*, the stable model semantics) collapse. Also notably, for the majority of cases (*e.g.*, all policies examined in this chapter), the two semantics coincide anyway. For the purposes of JustAct, it suffices to conflate *unknown* and *false* valuations

of facts: neither is true. For example,  $p \text{ if not } p$  formalises a logical contradiction, giving  $p$  unknown value; agents agree that  $p$  is not true.

3. **Facts** are inductively defined. As in  $\text{Datalog}^-$ , facts may be constructed by applying a constant predicate symbol to a list of constants or variables. However, unlike  $\text{Datalog}^-$ , (sub)facts may occur wherever variables may occur, and variables quantify over *ground facts*, and not only constant symbols. In database and logic programming literature, this feature is well studied, and is usually called *function symbols* [Llo84, RU95, CGMT15, GMT13]. The syntax of rules are generalised accordingly; consequents and antecedents are facts that are possibly composed from smaller facts. Variables may occur in facts at any depth. Naturally, the semantics is generalised to handle these composite facts. This generalisation lets Slick policies more easily model various other forms of policy, for example, policies expressed in eFLINT via nested eFLINT fact-instances. In general, this feature affords the more terse and intuitive definition of policies which are comparatively cumbersome and un-intuitive when expressed in  $\text{Datalog}^-$ . *E.g.*, consider how the Slick rule `some vote if Agent votes` is applicable regardless of the internal structure of the fact binding the variable `Agent`, which would otherwise require a unique  $\text{Datalog}^-$  rule specialised to each shape of  $\text{Datalog}^-$  fact to be bound by `Agent`.
4. **Fact arguments are homogenous** and parentheses are only needed to avoid structural ambiguity. For example, `eats(amy, apples)` in  $\text{Datalog}^-$  can be translated into Slick as `amy eats apples`, which is read very naturally. Firstly, because predicates are constants like any other, they can be abstracted by variables; consider how, in the prior example, only in Slick could constant `eats` be abstracted, *e.g.*, by variable `Uses`. Secondly, note how this reduces the visual complexity of rules, because one layer of parenthesis is avoided.
5. Slick supports **(in)equality constraints** over (all pairs in sets of) facts as antecedents, using keywords `same` and `diff`. This is a well-studied feature of logic programming languages, *e.g.*, described in [GM84]. In practice, it can tersely capture common patterns in agreements, where the applicability of key rules is conditioned on particular facts (mis)matching. For example, `error if X authorises and not same { X amy }` is applicable if `X` binds `bob`, but not if `X` binds `amy`. Note `not same { X Y Z }` and `diff { X Y Z }` have different meanings; only the former is satisfied in the case where `x = y ≠ z`.
6. As we have described it thus far ('ideal Slick'), the inference procedure underlying the Slick semantics is non-terminating. For example, consider attempting to finish enumerating the truths of `Fact is true if Fact. it is raining`. Unfortunately, ideal Slick belongs to the large subset of conceivable logic programming languages

for which checking termination is undecidable. The space of workarounds and compromises has been thoroughly researched for many years [dSD94]. As this is not the focus of this work, we opt for a pragmatic approach to ensuring that inference terminates: we **bound the maximum depth of true facts** to a statically-configured parameter, as in [Cho95]. However, where [Cho95] ignores facts deeper than the bound, we instead detect and make explicit when the bound is exceeded: each such policy is trivialised and invalidated; precisely, its denotation is replaced by that of the policy `error. bound exceeded`. Thus, for all policies, a) with enough computational resources<sup>3</sup>, queries are eventually answered, and b) validity guarantees that each logical implication encoded by each rule is satisfied by its truths.

7. We define validity as for Datalog<sup>⊥</sup> in Section 6.3.1: policy  $p$  is valid unless the fact `error` is true in  $p$ . Other than this special meaning, `error` is an entirely ordinary fact. Section 6.6.3 demonstrates, and Section 6.7 discusses, how Slick affords agents' *cooperative specification* of shared policies, via its *composition control* language features [EvB24]. In summary, agents agree that only *valid* justifications are useful, where validity captures requirements of the contributors to the policy, including requirements on the runtime context of the contributions themselves. Note that, by using features 3 and 4, we can introduce a degree of explainability by deriving verbose `error`-facts that refer to the rule deriving them; e.g., the program `error (b was true) if b. error if error X.` will implement validity as expected while its model informs the observer why it is invalid.

The following presents the Datalog<sup>⊥</sup> rules shown in Table 6.1 translated into Slick. Because Slick is closely related to Datalog<sup>⊥</sup>, the correspondence between the original and translated rules is evident, with only minor differences in their concrete syntax.

**Example 6.1** (Domain-Level Slick Rules: Specifying Conditions on Data Deletion).

```
error if    any confirms. any confirms if any trusts X.
error if not any confirms. any trusts bob.
bob deletes data1.
```

### 6.5.3 Connecting Policies to the Runtime System

We complete the instantiation of the JustAct framework by defining the remaining concepts in Figure 6.2. These final concepts inter-connect the policy language (in Section 6.5, currently) with the rest of the runtime system (in Section 6.4, previously).

<sup>3</sup>The Slick interpreter can be configured to add further restrictions to validity, to bound the resources needed for reasoning. For example, one can define the maximal number of truths.

### 6.5.3.1 Defining Effects and Affectors

Effects connect the (output of) the policy reasoning process to the behaviour of agents in the data exchange system: agents realise the effects of (justifications of) actions as the reading and writing of data in the data plane.

Recall from Figure 6.2 that effects themselves are a static concept: they are a subset of *truths* that statically identify their *affector* agents. We let each truth of a fact matching `Agent reads Data` or `Agent writes Data` be effects, and let their affectors be `Agent`. For example, without any runtime context, it is evident that the policy `(amy reads data1. amy is happy.)` has two truths, but only the effect `amy reads data1`, whose affector is `amy`. An effect is connected to a particular action, at runtime, when the effect is *enacted* (Definition 6.4). Agents can *realise* these effects by influencing the data plane in the corresponding manner. For example, once the effect `amy reads data1` is enacted, it can be realised by `Amy` reading the data identified by `data1`.

Because effects are the interface between the control- and data-plane, any definition of effects ultimately determines the granularity at which policies can regulate the data plane. By our definition, each effect identifies its affector and the data, but notably, it does not identify the *contents* of the data. Consequently, in this chapter, policies cannot distinguish the contents written or read by agents in the data plane.

### 6.5.3.2 Reflecting Policy Context via Extract

Finally, the function  $extract : message\ set \rightarrow policies$ , specified visually as the arrow  $message\ set \xrightarrow{extract} policies$  in Figure 6.2, yields the policy contents of any given (set of) messages: the input to the policy reasoning process. As was done in `Datalog-` in Section 6.3.1, our definition is such that the policy extracted from an individual message `Msg` reflects its runtime context. This is implemented in Rust as a function with the type signature `fn extract_from(payload: Policy, msg_id: GroundAtom) -> Policy`. Precisely, the result is a syntactic transformation of the arbitrary *payload*: for each rule, each existing consequent `Fact` is supplemented by the consequent `Fact within Msg`. Intuitively, messages can assert any truths, but not without tracing back to the message identifier (and the author). As in Section 6.3.1, *extract* is applicable to non-singleton message sets also, in which case the result is the composition of the policies extracted from each message independently. Finally, in the context of (the justification of) an action by actor `Agent`, the extracted policy undergoes a second syntactic transformation: the rule `actor Agent` is added.

Agents rely on *extract* to reflect the runtime context of a message in (a transformed version of) its payload. Notably, truths of the form `Fact within (Author M)` trace the truth of `Fact` to the message `Author M` authored by `Author`. In Section 6.6, to follow, these

features let the usual rules of Slick specify the powers of agents to make statements and take actions, or to (conditionally) *delegate* these powers to other agents. But for now, we demonstrate with a more familiar example: the Slick rules below encode the *ownership* mechanism, defined in natural language in Section 6.3.1, regulating the authorship of the rules above. Consequently, for example, rules asserting `amy confirms` are invalid unless they are authored by `amy`.

**Example 6.2** (Meta-Level Slick Rules: Specifying the Authors of Stated Rules).

```
error if      (amy confirms) within (Author M) and diff { Author amy }.
error if      (amy trusts X) within (Author M) and diff { Author bob }.
error if (bob deletes data1) within (Author M) and diff { Author bob }.
```

### 6.5.4 The Slick Interpreter Implementation

The Slick interpreter interprets the Slick language, as it is described in Section 6.5.2, yielding its *denotation*: a logical valuation of each fact. Hence, the denotation of each  $p$  also reflects the validity of  $p$ : is `error` true in  $p$ ?

The work of the interpreter can be divided into two stages: *parsing* and *inference*:

1. The input is **parsed** into an intermediate representation: a set of message-annotated rules. There are six kinds of antecedent, for the six combinations of the following two choices: a) positive or negative (`not`), and b) truth-check, equality-check (`same`), or inequality-check (`diff`). For example, `error if agent A and not same { A amy }` has only consequent `error`, a check that `agent A` is true, and a check that `A` and `amy` are not equal.
2. The interpreter implements the **inference** algorithm in [Prz90] as straightforwardly as possible. From the initially empty fact set  $S_{\text{false}}$ , the inference procedure computes new  $S_{\text{true}}$  fact set by exhaustively instantiating and applying the given finite rules. In case the depth of a fact exceeds the configured maximum bound, inference restarts with the input policy (`error. bound exceeded.`). Otherwise, the process continues, at each step initialising the next  $S_{\text{false}}$  as the complement of  $S_{\text{true}}$ , which can be understood as operationalising ‘assume everything we failed to prove is false’. Provably, this process continues to an alternating fixed point:  $S_{\text{true}}$  and  $S_{\text{false}}$  alternate forevermore. The final truths are  $S_{\text{true}}$  and the unknowns are  $S_{\text{false}} \setminus S_{\text{true}}$ . See [Prz90, VGRS91] for more details on the well-founded semantics, and the intuition behind the alternating fixpoint.

The Slick interpreter is implemented as a Rust library. Listing 6.5 shows the library’s API, *i.e.*, the `Policy` data type and its methods; users parse policies from text and then compose and inspect their denotations.

Listing 6.5: Simplified Rust source code of the Slick parser and interpreter.

```

type Policy = Vec<Rule>;
struct Rule {
    consequents: Vec<Atom>,
    pos_antecedents: Vec<Atom>,
    neg_antecedents: Vec<Atom>,
    checks: Vec<Check>,
}

struct Check {
    atoms: Vec<Atom>,
    kind: CheckKind,
    positive: bool,
}

enum CheckKind { Diff, Same }

enum Atom {
    Constant(Text),
    Tuple(Vec<Atom>),
    Variable(Text),
}

enum GroundAtom {
    Constant(Text),
    Tuple(Vec<GroundAtom>),
}

impl Policy {
    fn parse(text: &str) -> Option<Self>;
    fn denotes(self) -> Option<Denotation>;
}

struct Denotation {
    trues: Vec<GroundAtom>,
    unknowns: Vec<GroundAtom>,
}

impl Denotation {
    fn is_valid(&self) -> bool;
}

```

## 6.6 Case Study: Processing Distributed Medical Data

In this section, we demonstrate and evaluate the JustAct framework (Section 6.3), instantiated as a data exchange runtime system (Section 6.4) with the Slick policy language (Section 6.5), and then applied to policy-regulated medical data processing across institutional domain boundaries. Section 6.6.1 overviews the subject of our case study: the BRANE component of the EPI framework. Section 6.6.2 (re)presents BRANE through the lens of the JustAct framework, and initialises our runtime system (instantiating JustAct) and then Section 6.6.3 walks through its dynamic behaviour in application to five representative usage scenarios. Finally, Section 6.6.4 evaluates the characteristics of our runtime system in application to BRANE’s usage scenarios, drawing particular attention to the role of the JustAct framework.

### 6.6.1 The Case: The Brane Component of the EPI Framework

The *Enabling Personalised Interventions* (EPI)<sup>4</sup> project aimed to create digital twins in healthcare to improve patient treatment prediction [KAA<sup>+</sup>24]. Research on EPI investigated the challenges of safely coordinating the federated processing of medical data across organisational boundaries within the European Union, *e.g.*, between hospitals, universities, and research centres.

An overarching technical contribution of the EPI project is the *EPI Framework*, which is conceptualised as a modular suite of software components that work together to control and automate facets of data-processing pipelines [KdLTG20, KME<sup>+</sup>24]. The most complete and recent overview of the EPI framework is given in [KME<sup>+</sup>24].

<sup>4</sup><https://enablingpersonalizedinterventions.nl> lists the EPI project members and deliverables.

Various articles report on the development of its specialised components. For example, [KVBG21] (re)routes network messages between containerised functions to the satisfaction of security constraints.

In this work, we focus on BRANE, the component responsible for unfolding users' processing workflows by planning, orchestrating, and coordinating the execution of workflow tasks. Each task is executed by a *worker* service, creating new (medical) data from existing data. Workers and data are distributed over physical machines and (technical representations of) *organisational domains* [VCB21]. For example, a hospital is a domain that provides and controls some medical datasets.

[EMvBB22] focuses on the regulatory facet of BRANE: each domain is represented by a *checker* service that reasons about and enforces the domain's *policies*, asserting control over the usage of the assets provided by the domain. For example, via the checker, a human policy expert defines the hospital's policy, which expresses constraints on the permitted processing of the hospital's medical records in order to protect the privacy of patients. In this context, an extra complication is that domain-local policies express decisions in terms of *meta* data that may also be sensitive, e.g., 'the processing of data-asset `foot-fracture-x-rays1.png` requires the consent of Amy' reveals sensitive information about Amy. [EMvBB22] addresses this complication by prescribing a policy-worker interface that gives domains maximal control over the extent to which private policy information is publicised via their policy decisions. Precisely, the execution of planned execution steps awaits *authorisations* from the involved domains, just in time. Hence, domains control the usage of their data, and they also decide how to balance policy privacy against system productivity.

Below, we clarify the usage of BRANE by enumerating the kinds of its constituent services and overview their roles and interactions. They are listed in an order that suggests the progression of a successful workflow execution from top to bottom:

1. A **driver** defines computational workflows: directed acyclic diagrams expressing the functional dependencies between assets and computation tasks. Each result (data) depends on a containerised function (data) and arguments (data). The driver reads their workflow's results once they are available.
2. The **planner** *plans* workflows, assigning workflow tasks to workers, for execution.
3. A **checker** *authorises* workflow plans whose execution (and the necessary reading and writing of data by workers) conforms to the checker's local policies.
4. A **worker** executes assigned workflow tasks, once authorised. The worker reads the function and argument data, then computes and writes the result.

Of the above services, only the planner is generally centralised, ultimately orchestrating the other services' tasks by planning the assignment of tasks to workers. Each

organisational domain hosts drivers, checkers, and workers. These services represent the domain's interests by participating in the (regulation of) data processing.

In principle, any subset of services are automated. In the existing BRANE implementation, each is implemented as a complex sub-system, involving a mix of human user interaction and automation. For example, drivers are fully automated services, but each interfaces BRANE at large with a data scientist user, whose defines the workflows to be executed. Similarly, checker services participate in BRANE automatically, as guided by periodic input from domain-local policy experts.

## 6.6.2 Initialising our Runtime System for Brane Scenarios

Figure 6.6 visualises the refinement of our runtime system such that BRANE concepts are mapped to elements of our runtime system. First and foremost, we map BRANE's notion of (organisational) *domain* to our notion of *agent*, such that distinct domains are distinct agents. This determines the granularity at which control is distributed, and at which the autonomous entities are distinguished. Hence, each agent can *play the role of* several of BRANE's services. For simplicity, this abstracts away the internal organisation of each domain. For example, we do not consider contention over several human users' and several services' to make statements and take action as agent *st-antoni*, which represents the organisational domain of the St. Antonius hospital. Throughout this section, we use St. Antonius (a Dutch hospital) and SURF (a Dutch National Research & Education Network (NREN) services provider) as examples of domains. These are real contributors to the EPI project. But our example users (Amy, Bob, and Dan) are fictitious.

In Sections 6.6.2.1 and 6.6.2.2, to follow, we complete the mapping of BRANE concepts to facets isolated to the data and control planes, respectively.

### 6.6.2.1 The Brane Data Plane

Our runtime system and BRANE rely on a similar abstraction: there exists a shared (initially empty) store of asset data, which is read and written by the agents.

Agents' access to data in the data plane is regulated via the connection to enacted effects, as specified previously in Section 6.6.2.1. Any agent *Agent* can introduce fresh data *Data* using the existing mechanism of executing workflow tasks: it suffices for *Agent* to define a fresh workflow (playing the role of driver) with no inputs and the output *Data*, and executing the task (playing the role of worker). In the next section, we design the policies such *Agent* is always permitted to enact the above, and it has the desired effect: *Agent* writes *Data*.

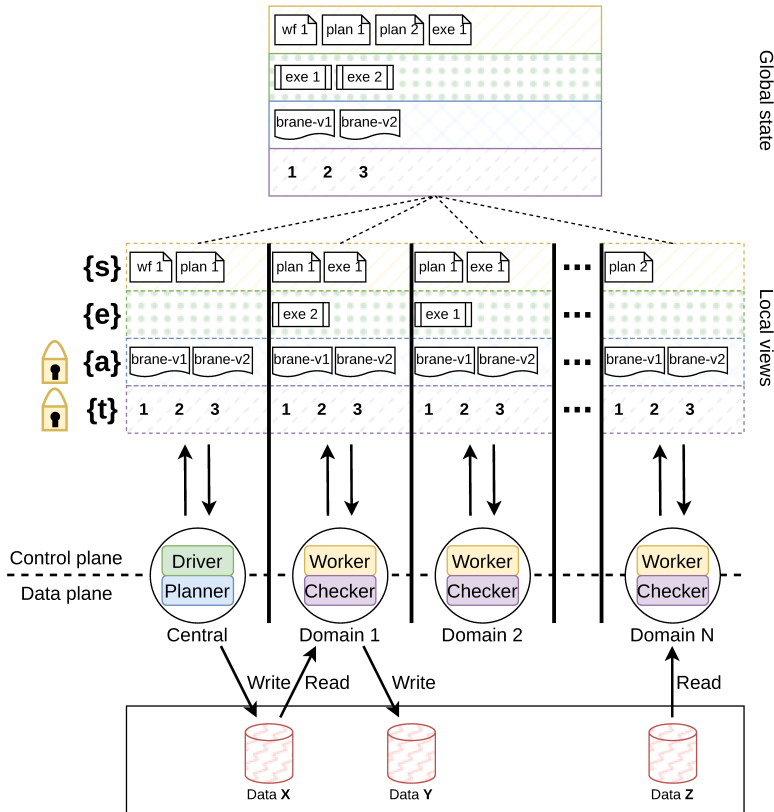


Figure 6.6: Graphical depiction of our runtime system instantiated for application to BRANE usage scenarios. Hence, this figure refines Figure 6.4: BRANE services map to our agents, and (the distinct kinds of) BRANE control meta data are mapped to meta data in our control plane. BRANE’s medical data maps trivially to our data plane.

### 6.6.2.2 The Brane Control Plane

In the initialisation of the control plane of our runtime system, we reproduce BRANE’s tracking and control of agents, assets, workflows, and so on.

In the original BRANE, system properties are defined in publications such as [KME<sup>+</sup>24], for users to reason about. However, naturally, there is a gap between these concepts and their encoding in the BRANE implementation, which are ultimately enforced at runtime. For example, [KME<sup>+</sup>24] uses precise wording and logical notation to define the obligation of workers to observe authorisations from involved checkers.

In our version of BRANE, we express these properties via policies in statements and agreements. Thus, we hope to narrow the gap between the high-level specification of these properties, and their encoding in a form that is systematically enforced: these coincide in Slick policy rules. Precisely, many properties of BRANE are specified in

Agreement 1: the *initial agreement*, representing the initial consensus on the basis on all actions at the (only) initial time 1. Precisely, until the current agreements are changed, permitted agreements are necessarily based on Statement consortium 1. The author of this statement is the *consortium*, which corresponds to no particular physical entity or human user, but represents the consensus of the other agents.

**Agreement 1: Statement consortium 1 is agreed at time 1.**

Statement consortium 1 lays out the payload policy of the initial agreement. Its concepts are stated as an indivisible unit, such that none of its rules can be used without the others. However, we present its contents broken up into several parts; to follow, we discuss each part in turn, and the facet of BRANE it encodes.

Figure 6.7 visualises the ontology encoded in Statement consortium 1 in the same style as Figure 6.2. This visual overview is included to aid the reader’s interpretation of the Slick rules comprising the policy itself. Statement consortium 1 encodes the shown ontological sets as facts matching structural patterns, e.g., *inputs* are encoded as facts matching `Task has input Variable`. Ontological functions are encoded either structurally, e.g., a sub-fact of each *input* is a *task*, or they are encoded in truth via rules, e.g., `ready  $\Leftarrow$  executed` is encoded as `error if Task executed and not Task ready`. Other policy rules layer additional meaning atop the ontology. Notably, via rules conditioning invalidity, it specifies the inter-agent power dynamics. For example, only the driver of a task may define its inputs. A detailed breakdown of Statement consortium 1 follows.

**Statement consortium 1 (the initial agreement statement).**

```

----- Part 1 -----
error if (Fact within Msg1) within Msg2.
error if (actor Agent) within Msg2.

Sayer says Fact if Fact within (Sayer M) and diff { (consortium 1) (Sayer M) }.
error if Sayer says (Agent says Fact).

----- Part 2 -----
Sayer drives Task if Sayer says (Task ready).
Sayer drives Task if Sayer says (Task has input Variable).

```

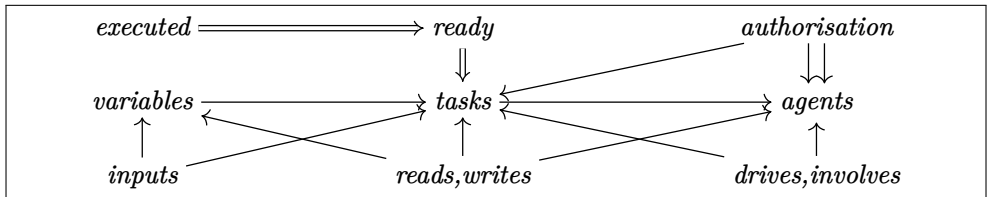


Figure 6.7: Graphical ontology of Agreement 1, which formalises essential requirements of BRANE. Sets (italicised) are related by pure, total functions ( $\rightarrow$  and  $\Rightarrow$  arrows) from domain to co-domain. Each ( $\Rightarrow$ ) denotes an identity function, e.g., `ready  $\subseteq$  tasks`.

```

Sayer drives Task if Sayer says (Task has output Label).

error if Sayer says (Agent drives Task).
error if Agent drives (Driver Name) and diff { Agent Driver }.

----- Part 3 -----
error if Task executed and not Task ready.
error if Task2 executed and Task2 has input (Task1 Label) and not Task1 executed.

----- Part 4 -----
Worker reads Variable if Task has input Variable and actor Worker
and Worker says (Task executed).
Worker writes (Task Label) if Task has output Label and actor Worker
and Worker says (Task executed).
error if Sayer says (Worker reads Variable).
error if Sayer says (Worker writes Variable).

----- Part 5 -----
error if Worker says (Task executed) and (Task has input Variable) within Msg
and Task involves Checker and not Checker says (authorise Task in Msg by Worker).

error if authorise Task in Msg1 by Worker
and (Task has input Variable) within Msg2 and diff { Msg1 Msg2 }.

----- Part 6 -----
Task involves Checker if Checker controls (Task Label).
Task2 involves Checker if Task2 has input (Task1 Label) and Task1 involves Checker.

```

Part 1 of Statement consortium 1 creates and protects basic abstractions that truths are intended to provide over the runtime context of policies. Recall that the policy extracted from a message is a transformation of its payload that asserts the following two kinds of property: 1. if a rule within message `Msg` has consequent `Fact`, then it also has consequent `Fact within Msg`, and 2. in the context of an action with actor `Agent`, rule actor `Agent` is added. The first two rules, respectively, preserve the converse properties, *i.e.*, preventing (the authors of) statements from mimicking the reflection mechanism with their own rules. This immediately demonstrates our first of many usages of Slick’s *composition control features* [EvB24]; Slick rules specify which other rules may be included, as a function of their runtime context. The other two rules then create and likewise protect a new abstraction: each truth of the form `Sayer says Fact` reflects the author of the rule inferring `Fact`. This relies on Definition 6.5: each message identifier matches `Author X`, where `Author` identifies the message author.

The condition `diff { (consortium 1) (Sayer M) }` in Part 1 gives special status to the rules within Statement consortium 1, for one reason, to avoid the inference of infinitely large facts of the form `consortium says (consortium says ( ... ))`. In the remainder of Statement consortium 1, which encodes more concrete facets of BRANE, rules of the form `error if Agent says ( ... )` are included to protect many such abstractions.

Part 2 of Statement `consortium 1` characterises *drivers* as agents that 1. define the input-output dependencies of *tasks* on data, and 2. mark tasks as ready for execution. For example, a user defines a workflow via a typical scripting language (e.g., BraneScript, typically used with BRANE [VCB21]), including the assignment statement  $y := f(x)$ . The corresponding driver encodes this as a fresh task  $t$  by stating  $t$  has input  $(t\ f)$ .  $t$  has input  $(t\ x)$ .  $t$  has output  $y$ .

Figure 6.7 expresses the requirement that each task has a unique driver via the function  $driver : tasks \rightarrow agents$ , which is visualised as the arrow  $tasks \xrightarrow{driver} agents$ . As in Definition 6.5, we use the ‘prefix trick’ to encode this requirement in Part 2 of the initial agreement: each task is a driver-*name* pair, where names discriminate the tasks with the same driver; i.e., we use drivers as task namespaces. Hence, we reduce the (complex) problem of disambiguating the driver of a task to the (simple) problem of naming tasks. This problem is simple in BRANE, as drivers already have distinct identifiers. The first four rules defines how agents behave as drivers, and permits only agents identified as drivers (via the aforementioned prefix trick) to act as drivers. The final rule ensures that only the driver identified by the task drives the task.

Part 3 prevents the premature execution of tasks. Precisely, tasks cannot be executed until they are ready, and the tasks producing their inputs are also executed. The next part defines the *effects* of enacting the execution of tasks. We again use the ‘prefix trick’ to translate  $output\ of : variables \rightarrow tasks$ , which is visualised as the arrow  $output\ of \xrightarrow{} tasks$  in Figure 6.7, into the initial agreement: each variable is a task-*label* pair, where labels discriminate the variables of the same task; i.e., variables globally identify asset data, while labels are task-local.

In Part 4, the first two rules infer the *enacted effects* of actions enacting the execution of a task: the acting worker reads each input, and writes each output as truths matching `Worker reads Variable` and `Worker writes Variable`. The rest of Part 4 prevents these effects from arising in any other way. Recall from Section 6.4.2 that the runtime system uses these effects to regulate agents’ access to asset data; agents can only read or write asset data when these are the effects of their (permitted and taken) actions. The scenarios will demonstrate several examples of agents reasoning backward from the effects they desire to the necessary actions and statements.

Part 5 of Statement `consortium 1` encodes the roles of checkers in BRANE. In summary, the execution of planned workflow tasks is conditioned on the authorisation of all *involved* agents. The final rule keeps authorisers from unintentionally authorising tasks with unexpected inputs. Precisely, agents have a means of asserting that the given message *closes the inputs* (Definition 6.11) of some task  $t$ ; each authorisation is scoped to a particular message, and the policy is invalidated if any other message attributes input variables to the authorised task. Consider how drivers cannot be

prevented from defining  $t$  differently in different statements. However, this rule prevents workers from executing  $t$  with some input  $v$  (from one message) as justified by an authorisation without  $v$  (from another message), unintentionally or otherwise.

**Definition 6.11** (messages  $M$  close the inputs of task  $t$ ).  $closeInputs(M, t) \triangleq$   
 $\forall v \in facts, M' \supseteq M, \neg true(extract(M), t \text{ has input } v) \wedge$   
 $true(extract(M'), t \text{ has input } v) \rightarrow \neg valid(extract(M')).$

Finally, Part 6 reproduces BRANE’s inductive definition of involvement from [KME<sup>+</sup>24]. In the base case, agents are involved with data they control. In the inductive case, a checker’s involvement propagates over tasks from their outputs to their inputs, in the direction of data flow. Intuitively, checkers are involved with all derivatives of the data under their control. This mechanism gives each involved checker the power to veto the task’s execution by withholding their authorisation, *e.g.*, reflecting their policy decisions without otherwise revealing the domain policies themselves [EMvBB22]. For example, a hospital authorises a particular trusted worker to process their sensitive medical records, but vetos the result being further processed elsewhere by withholding any further authorisations.

### 6.6.3 Usage Scenarios

Here we walk through a sequence of scenarios in which we apply our initialised runtime system to new and existing usage scenarios of BRANE. Note that our discussion of the first scenario is the most detailed, because most of its details are novel.

#### 6.6.3.1 Existing Usage: Isolated Authorisation and Execution

We demonstrate the application of our system to a representative usage of the original BRANE system: several agents cooperate to drive the execution of data-processing workflows, as permitted by agent reasoning in isolation about their local policies. In our scenario, this isolation manifests as truths of the form `authorise Task in Msg by Worker` being stated only as facts, *i.e.*, inferred by variable- and condition-free rules. Effectively, agents entirely externalise their reasoning about local policies, and reveal only the resulting authorisations, on a case-by-case basis.

The scenario begins with SURF creating fresh data by executing a zero-input task in the statement `Statement surf 1`. The data is bound to the fresh variable `(surf utils) entry-count`, which can be used as input in other workflows. Figure 6.8 depicts these, and all (relations between) tasks and variables used in Section 6.6.3.

**Statement surf 1** (SURF provides a utility asset).

`(surf utils) has output entry-count.`

```
(surf utils) ready.
(surf utils) executed.
```

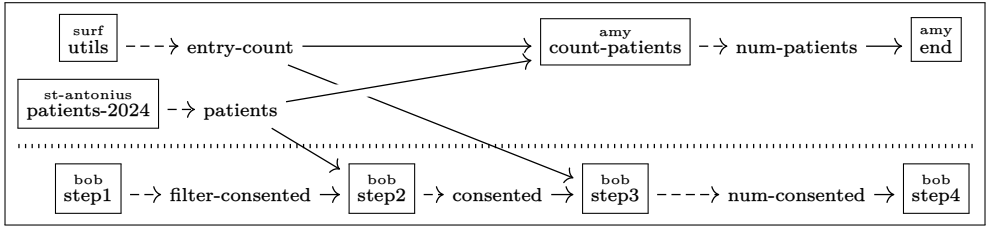


Figure 6.8: Bipartite graph of workflow tasks (*task-names* overset by *-drivers* in boxed vertices) and asset variables (*variable-labels* in unboxed vertices), in scenarios of Section 6.6.3, related by data-dependencies in the direction of data flow; output (dashed arrows) relates tasks to variables, and input (solid arrows) relates variables to tasks. The dashed line separates tasks defined in Scenarios 1 (above) and 2 (below).

We let ‘Analyst’ Amy drive the completion of a one-task medical data workflow. Amy is a (human) data scientist working to collect basic statistics of European hospitals, starting with the St. Antonius hospital. Amy defines a fresh task with the suggestive name `count-patients`, and marks it ready for execution. Intuitively, it encodes an assignment that would be rendered as  $numPatients := entryCount(patients)$  in typical imperative workflow languages. Input variables with labels `entry-count` and `patients` are outputs of tasks driven by `surf` and `st-antoni...`, respectively. Amy publicises this statement to her peers.

On its own, Statement `amy 1` defines a task, but does not justify its execution. Formally, there exists no valid subset of the statements we have shown so far that include Statement `amy 1`. Intuitively, the execution of `st-antoni... patients-2024` has no valid justification, but Amy’s task depends on its output.

**Statement `amy 1`** (Amy prepares a patient-counting task).

```
(amy count-patients) has input ((surf      utils      ) entry-count).
(amy count-patients) has input ((st-antoni... patients-2024) patients ).
(amy count-patients) has output num-patients.
(amy count-patients) ready.
```

After some time, Amy acquires Statement `st-antoni... 1` from St. Antonius, and propagates it to the workers in the system. Whether this statement was made before, or in reaction to Amy’s statement is not important; it suffices that all its observers agree that it was authored by St. Antonius. Note that it includes an assertion that St. Antonius controls the `patients` data, so tasks cannot process it (or its derivatives) without the authorisation of St. Antonius. This authorisation is given by St. Antonius to themselves for task `st-antoni... patients-2024`, but not yet for tasks processing the outputs of `st-antoni... patients-2024`, for example, task `amy count-patients`.

**Statement** `st-antoni` 1 (St. Antonius asserts control of the patient data).

```
(st-antoni patients-2024) has output patients.  
(st-antoni patients-2024) ready.  
st-antoni controls ((st-antoni patients-2024) patients).  
  
(st-antoni patients-2024) executed.  
authorise (st-antoni patients-2024) in (st-antoni 1) by st-antoni.
```

‘Disruptor’ Dan observes these statements. Inspired by Statement `st-antoni` 1, Dan attempts to establish control over `patients` by stating Statement `dan` 1. All observers agree that this statement is valid, and, as no action is taken, Dan remains well-behaved. Indeed, it is possible that, after some more statements, actions may be justified using Statement `dan` 1. However, Dan has misunderstood his role in the system. The crucial difference between Dan and St. Antonius is the incentives they provide for agents to accept their conditions of control over `patients`. Only St. Antonius has coupled the condition with useful rules, namely, the definition and execution of task `st-antoni patients-2024`. In contrast, there is no incentive to include Statement `dan` 1 in justifications, so the statement goes unused. Agents who expect this outcome can exercise their right to forget Statement `dan` 1 altogether. There also exists no statement (for example, by Dan) that causes confusion between variables `st-antoni patients` and `dan patients`.

**Statement** `dan` 1 (Dan attempts to control the patient dataset).

```
dan controls ((st-antoni patients-2024) patients).
```

SURF states their own execution of Amy’s task. On its own, Statement `surf` 2 is invalid; it appears to execute a task that is not ready! Furthermore, the union of Statement `surf` 1 with any subset of the prior statements is invalid, for example, because it lacks the necessary authorisation of St. Antonius. However, this is not a problem; indeed, agents are expected to sometimes make invalid statements, because these ultimately enable the creation of valid justifications later. In this case, the statement enables actions justified by SURF executing Amy’s task `amy count-patients`. Intuitively, SURF lets St. Antonius decide who executes the task. It all depends on whose execution of the task St. Antonius chooses to authorise.

**Statement** `surf` 2 (SURF executes task `amy count-patients`).

```
(amy count-patients) executed.
```

Externally, St. Antonius reasons that they prefer to execute this task themselves, as it requires little work, yet handles extremely sensitive data. They make Statement `st-antoni` 2. Shortly thereafter, St. Antonius takes a new action at time 1, with basis Statement `consortium` 1, and a justification consisting of Statements {`consortium` 1,

amy 1, surf 1, st-antoni<sup>s</sup> 1, st-antoni<sup>s</sup> 2 }, *i.e.*, including all statements so far except {dan 1, surf 2}. All observers agree that this is permitted, by Definition 6.3:

1. Each message in the justification is **stated**.
2. The basis (Statement `consortium 1`) is a current agreement. In fact, it is the only one. Recall Property 6.3: agents always agree on the current agreements.
3. The policy extracted from the justification is **valid**, because it falsifies `error`. Every observer can verify this independently by extracting the policy from the provided justification, and evaluating the truth of `error`. Because Slick's inference procedure is deterministic, observers necessarily reproduce this result.
4. The action is **current**, because its basis applies at the current time.

Consequently, all observers agree that this action preserves the good behaviour of St. Antonius. This includes auditors arbitrarily far into the future, as this action is logged alongside sufficient contextual information (*e.g.*, the current time was 1) to check permission. All observers also agree on the effects of the action: St. Antonius reads the entry-count and patient assets, and writes the count asset. Soon afterward, St. Antonius uses this action to mirror these effects in the data plane by reading and writing the asset data assigned to these variables.

**Statement** `st-antonis 2` (St. Antonius authorises Amy's task).

```
authorise (amy count-patients) in (amy 1) by st-antonis.  
((amy count-patients) num-patients) executed.
```

Amy reasons backwards from the ultimately desired effect: Amy aims to read the patient count, which is encoded in Slick as `amy reads ((amy count-patients) num-patients)`. Amy's conclusions are captured in Statement `amy 2`; Amy plans a final task which has (only) the count as input, and thus, will have the desired effect when enacted.

Amy has the physical capability to enact this statement already, but not without observers agreeing that the action is not permitted! Amy must include Statement `st-antonis 2` in the justification (to witness the execution of the input), but then Amy must include Statement `st-antonis 1` (for the same reason), but then Amy must provide evidence that St. Antonius authorises Amy executing the new task. No existing statement includes such an authorisation, and Amy cannot state it herself (while preserving validity). Amy chooses to stay well-behaved by not acting just yet.

**Statement** `amy 2` (Amy reads the workflow result).

```
(amy end) has input ((amy count-patients) num-patients).  
(amy end) ready.  
(amy end) executed.
```

Upon observing Amy’s new statement, St. Antonius is driven to the obvious conclusion; Amy awaits the authorisation to act. St. Antonius reasons that stating this authorisation would enable Amy enacting the execution of the task, with the effect of Amy reading count, which is a derivative of patients. However, St. Antonius concludes that this derivative of the sensitive data it not sensitive itself, and so there is no harm in Amy reading it. All of this reasoning is hidden from external observers, but its result is publicised as Statement `st-antonius 3`.

**Statement `st-antonius 3`** (St. Antonius lets Amy read the derived asset).

```
authorise (amy end) in (amy 2) by amy.
```

Upon observing this new statement, Amy takes a new action at time 1, as justified by all statements so far, except for Statements `surf 2` and `dan 1`. Once again, all observers conclude that this action is permitted, and so, it preserves the good behaviour of Amy. All observers agree that this action has the (intended) effect `amy reads num-patients`. Amy externalises this effect by reading the external asset that corresponds to the count variable.

### 6.6.3.2 Existing Usage: Distributed Workflow Execution

Like the original BRANE system, our system affords the execution of multi-step workflows over multiple domains.

In this scenario, ‘Bookkeeper’ Bob wants to know how many patients at St. Antonius have consented to the processing of their patient data being read. To achieve this goal, Bob plays the role of another data scientist; Bob encodes the four-task workflow in the following statement. Bob also plays the role of a software provider in Step 1 by providing the `filter-consented` function to be used in Step 2.

The first part of Statement `bob 1` defines new tasks similarly to the previous examples. But Bob also includes additional rules in this statement. Firstly, a rule that asserts the need to plan the execution of all steps beforehand, via a straightforward rule. By including them in one statement, Bob’s task definitions are coupled with Bob’s condition on their execution. It also demonstrates that all agents (not just `consortium`) enjoy the expressive power of conditional rules to assert complex conditional truths. Bob also includes a blanket authorisation of agents to execute any task at all; Bob is not concerned with – nor responsible for – the sensitivity of any data. Finally, Bob plans to execute Steps 1 and 4 himself.

**Statement `bob 1`** (Bob defines and partially executes a workflow).

----- *Workflow Definition Part* -----

```
(bob step1) has output filter-consented.
(bob step1) ready.
```

```

(bob step2) has input ((bob step1) filter-consented).
(bob step2) has input ((st-antoni-us patients-2024) patients).
(bob step2) has output consented.
(bob step2) ready.

(bob step3) has input ((surf utils) entry-count).
(bob step3) has input ((bob step2) consented).
(bob step3) has output num-consented.
(bob step3) ready.

(bob step4) has input ((bob step3) num-consented).
(bob step4) ready.

```

----- *Partial Execution Part* -----

```

error if (bob Name1) ready and not (bob Name1) executed and (bob Name2) executed.

authorise Task in Msg by Worker
  if Worker says (Task executed) and (Task ready) within Msg.

(bob step1) executed.
(bob step4) executed.

```

St. Antonius and SURF observe and reason about Bob's statement. Externally to the system, they exchange communications, negotiating their possible roles in facilitating the execution. For example, SURF observes that Bob's statement implies Bob's authorisation to any executed task, but this is harmless, as it remains useless for tasks not involving Bob. While St. Antonius is willing to execute Step 3, they are unwilling to execute Step 2, as this requires them to execute Bob's untrusted `filter-consented` with the hospital infrastructure. Fortunately, St. Antonius is willing to let SURF execute Step 2 instead; St. Antonius trusts SURF to read the sensitive patient data, while SURF accepts the risk of executing Bob's mysterious function. None of this reasoning is revealed to the other agents, but its results are; SURF and St. Antonius make the following two statements, respectively.

**Statement** surf 3 (SURF executes step 2).

```
(bob step2) executed.
```

**Statement** st-antoni-us 4 (St. Antonius authorises and executes some of Bob's tasks).

```

(bob step3) executed.
authorise (bob step2) in (bob 1) by surf.
authorise (bob step3) in (bob 1) by st-antoni-us.
authorise (bob step4) in (bob 1) by bob.

```

At this point, the execution of all of Bob's tasks are justifiable (while preserving validity). Independently, Bob, SURF, and St. Antonius can identify and enact such a justification, *e.g.*, if Bob acts first, Bob can gossip the action to SURF, informing

SURF of which justification to use themselves! Each observer agrees on the permission and effects of each of these actions.

While their data-dependencies are *modelled* by their statements as inputs, outputs, reads, and writes, they are not enforced by policy; for example, the above statements suffice to let SURF enact Step 3, even before St. Antonius enacts Step 2. However, of course, these actions are also constrained by their data-dependencies in the data plane, so St. Antonius cannot execute Step 3 until the effects of executing Step 2 are externalised. However, this does not concern norms, justifications, or permissions, so they are not enforced via policies.

### 6.6.3.3 Existing Usage: Concurrent Driving, Execution, and Authorisation

In this section, we briefly remark on the observation that our system faithfully reproduces the isolation of independent workflows, *i.e.*, the definition, authorisation, and execution of different workflows proceed concurrently. Concurrent tasks are clearly visible in Figure 6.8 as each task-pair where neither depends on the other, *e.g.*, `amy end` and `bob step2`. But moreover, unrelated statements are created concurrently also. For example, all the activities of Scenarios 1 and 2 (in Sections 6.6.3.1 and 6.6.3.2, respectively) occur concurrently after `patients` is available. For example, if Amy crashes immediately after initialisation, and then stays entirely unresponsive, Scenario 1 cannot complete, but Scenario 2 will still complete successfully.

### 6.6.3.4 New Usage: Inter-Domain Policies

Thus far, our scenarios have faithfully reproduced a characteristic of the original BRANE; domain-local policies are entirely private: hidden from other agents, exposed only in the form of particular authorisations, publicised only when needed, on a case-by-case basis. Indeed, this is an important feature of BRANE, which was intentionally developed to handle the eventuality that domain-local policies often reflect private information, thus becoming private themselves [EMvBB22]. However, the original BRANE offers no way for domains to share policy information. For example, while St. Antonius can send information to SURF via channels outside of BRANE, it is disconnected from SURF's authorisations. In contrast, this scenario demonstrates how our system lets agents cooperate in defining composite Slick policies, leading to authorisation. In general, statements are useful for unifying the *informative* and *regulatory* qualities of policy. In the case of BRANE, statements can publish non-sensitive parts of domain-local policies; recipients of these statements benefit from having insight into the reasoning of the author, while the author benefits from being decoupled from (the work of) justifying other agents' actions.

From Section 6.6.3.1, it is clear that St. Antonius already models and reasons about the domain trustworthiness, which ultimately informs which tasks they choose to authorise. With the following statement, some of this reasoning is internalised in Statement `st-antonius 5` as conditional authorisations. Consequently, any observer can predict the authorisation of St. Antonius, and under the right conditions, infer it on their behalf. Moreover, the `x is trusted` relation is expressed in a form that other agents can use, in their own way, and for their own justifications!

**Statement** `st-antonius 5` (St. Antonius authorises executions by trusted workers).

```
st-antonius is highly trusted. surf is highly trusted.
```

```
Agent is trusted if Sayer says (Agent is highly trusted)
    and st-antonius says (Sayer is highly trusted).
```

```
authorise Task in Msg by Worker if (Task ready) within Msg
    and st-antonius says (Worker is trusted).
```

Given Statement `st-antonius 5`, SURF defines, readies, and executes the following task, which reads the patient data. Most interestingly, although this task is defined to involve St. Antonius, their participation in the action is not required at any point. In this sense, Statement `st-antonius 5` delegates a limited power to authorise on the behalf of the St. Antonius to SURF, despite this requiring SURF to reason about a publicised part of the local policy of St. Antonius.

**Statement** `surf 4` (SURF creates and executes a task).

```
(surf read-patients) has input ((st-antonius patients-2024) patients).
```

```
(surf read-patients) ready.
```

```
(surf read-patients) executed.
```

Next St. Antonius makes Statement `st-antonius 6`. Like Statement `st-antonius 5` before, this new statement internalises a facet of the domain-local policy (reasoning) of St. Antonius, to the benefit of letting others systematically reason on their behalf. In this case, Statement `st-antonius 6` captures the link between (simplified) consent of patients to the processing of their data, and the authorisation by St. Antonius of tasks processing patient data. The JustAct framework affords this usage; for example, this statement enables a fruitful cooperation between St. Antonius, AMdEX, and SURF, which are all (highly) trusted by St. Antonius.

However, unlike Statement `st-antonius 5`, this statement captures sensitive policy information; St. Antonius does not wish to disclose the set of patients to *untrusted* agents! Ideally, statements could be used to regulate such communications via justification and permission, as usual. Unfortunately, JustAct does not (yet) offer a means for agents to internalise this. For example, if SURF forwards Statement `st-antonius 6` to Dan, SURF still satisfies the definition of well-behaved.

**Statement** `st-antoni` 6 (St. Antonius authorises tasks with patient consent).

```
amy is a patient. bob is a patient. dan is a patient.
```

```
trusted consent for Task in Msg if st-antoni says (Sayer is trusted)
and Sayer says (Patient consents to Task in Msg).
```

```
Task lacks trusted consent if authorise Task in Msg by Worker via consent
and Patient is a patient and not trusted consent for Task in Msg.
```

```
authorise Task in Msg by Worker if Patient consents to Task in Msg
and Worker says (Task executed) and not Task lacks trusted consent.
```

### 6.6.3.5 New Usage: Dynamic Amendment of the Agreement

So far, all actions have been controlled via the concepts defined in the initial agreement. In this final scenario, we demonstrate more fundamental changes agents are able to make to the agreement in order to alter which actions are justifiable in the future.

After some negotiations, the agents agree that the initial agreement has some flaws. Namely, 1. St. Antonius cannot express their involvement in the sensitive `patients` without being involved in its insensitive derivative `count`, and 2. SURF lacks control over `entry-count`; they regret providing the asset in Statement `surf 1` unconditionally.

By cooperating (externally), the agents formulate Statement `consortium 2` as an altered version of Statement `consortium 1`: an existing part is altered, and a new part is added, respectively addressing the aforementioned flaws:

1. In the altered Part 6, the rules defining the *involvement* relation between tasks and checkers is restricted, such that the variable in question is not stated to be *insensitive*, which is a novel predicate over asset variables.
2. In the added Part 7, SURF's takes control over the `entry-count` asset.

**Statement** `consortium 2` (Statement `consortium 1` adjusted).

... Parts 1 to 5 of Statement `consortium 1` are repeated here ...

----- *Altered Part 6* -----

```
Task involves Agent if Agent controls (Task Label)
and not Agent says ((Task Label) is insensitive).
Task2 involves Agent if Task has input (Task1 Label) and Task1 involves Agent
and not Agent says ((Task Label) is insensitive).
```

----- *Added Part 7* -----

```
(surf utils) involves surf.
```

These changes are enforced: Agreement 2 is created, and the current time is advanced to 2. The actions at time 1 are unaffected, but until the current agreements are changed again, all actions must be based on Agreement 2.

**Agreement 2: Statement consortium 2 is agreed at time 2.**

St. Antonius takes advantage of the new notion of insensitive variables by making the following statement. Agents can then justify processing (derivatives of) `count` without the authorisation of St. Antonius, but authorisation is required to process `patients` as before. SURF is also pleased, because they have regained control of `entry-count`, letting them judge each execution before it happens, for example, to avoid it being used for any nefarious purposes.

**Statement st-antonius 7** (St. Antonius releases control over an asset).

`((amy count-patients) count) is insensitive.`

Some time later, an administrator of the system becomes suspicious that sensitive asset data has somehow leaked to ‘Disruptor’ Dan, via their exploitation of a vulnerability of the external asset store. The administrator exploits their power to ‘break the glass’, seizing emergency powers to regain control of the situation; all actions must be paused to prevent more reading and writing events, until the administrator completes their diagnosis. This is achieved by the administrator synchronously updating the current time to 3. Consequently, until an agreement applies, well-behaved agents cannot act!

Some time later, the administrator determines that it was a false alarm; Dan did not do anything wrong (yet). The productivity of the system is restored simply by restoring the prior agreement. Precisely, the new Agreement 3 is created, whose statement is the prior Statement `consortium 2`.

**Agreement 3: Statement consortium 2 is agreed at time 3.**

## 6.6.4 Evaluation of the Case Study

We conclude the case study by evaluating our implementation in application to BRANE’s usage scenarios.

Firstly, we briefly report on the empirical facet of our evaluation of our case study: the performance of our runtime system in each experimental scenario. Table 6.2 presents our measurements of the run duration for each experimental usage scenario from beginning to end (but after the runtime system and Slick interpreter are compiled). Note that, as our prototype implementation executes in shared memory, these results under-represent the overhead of (de)serialisation and network communication that would be present in realistically-distributed data exchange systems (see Section 6.4.1). Nevertheless, these results faithfully represent the work of the runtime system itself: (round-robin) scheduling of agents, logical routing of control messages, and logging of agents interaction with (meta) data. It also faithfully represents the work of agents: observing and reacting to their environment, then collecting,

composing, interpreting, and evaluating the denotations of received Slick policies, before creating and gossiping statements and actions to their peers.

Scenario	Section	Title	Duration
1	Section 6.6.3.1	Isolated Execution	$1260 \pm 180\mu s$
2	Section 6.6.3.2	Distributed Execution	$910 \pm 10\mu s$
3	Section 6.6.3.3	Concurrency <sup>†</sup>	$2040 \pm 510\mu s$
4	Section 6.6.3.4	Inter-Domain Policies	$1000 \pm 10\mu s$
5	Section 6.6.3.5	Dynamic Agreement	$800 \pm 10\mu s$

Table 6.2: Duration of our experimental data exchange system applied to the five BRANE usage scenarios, each reported as the mean and standard deviation of 10 runs.

<sup>†</sup>We measure the case where Amy does not crash, because otherwise, the scenario has no defined end.

In the remainder, we present the analytical facet of our evaluation of our case study. To structure the discussion, Table 6.3 enumerates the *desirable characteristics* of BRANE-like systems, distinguishing those already (E)xisting in BRANE from those that are (N)ew to BRANE, *i.e.*, present only in our own version of BRANE. In the discussion to follow, we explicitly connect discussion points to particular desirable characteristics, along with whether each point contributes positively (+) or negatively (−).

$E_1$	Data scientists can define workflows and read the results of their execution.
$E_2$	Users can introduce persistent assets for use as inputs to workflows.
$E_3$	The usage of assets is regulated by domain-local policies.
$E_4$	Checkers control the exposure of their domain-local policies to other agents.
$E_5$	Agents work unimpeded by other agents (not) doing unrelated work.
$E_6$	Agents straightforwardly reason about their roles in the system.
$N_1$	Checkers can share and delegate control over processing via policies.
$N_2$	Inter-agent power dynamics are enforced via user-facing domain-local policies.
$N_3$	Deviations from policies that cannot be prevented are detected by auditors.
$N_4$	Agents can robustly reason about and prove the permissibility of their actions.
$N_5$	Policies can be easily and extensively refined and changed at runtime.

Table 6.3: Desirable characteristics in BRANE-like systems which already (E)xist in BRANE, or which would be (N)ew to BRANE, to structure our qualitative evaluation.

Sections 6.6.3.1 to 6.6.3.3 focused on the reproduction of existing BRANE functionality. Section 6.6.3.1 demonstrated defining tasks let ‘Analyst’ Amy, St. Antonius, and SURF each define computational workflow tasks in their own statements. This lets Amy play the role of a workflow provider, ultimately reading the result of executing the workflow (+ $E_1$ ). It also lets SURF and St. Antonius play the role of asset providers, as their task outputs were inputs to Amy’s workflow (+ $E_2$ ). In the case of St. Antonius,

patient data was provided along with an assertion of control over the processing of the asset and its derivatives; hence, workers could not justify executing Amy’s workflow tasks without authorisation from St. Antonius (+ $E_3$ ). Actions interfaced with the domain-local policies of their controllers via the latter’s explicit authorisations, which publicised only their policy-decisions themselves (+ $E_4$ ). In fact, as in [EMvBB22], domains could intentionally postpone their authorisations, or mask the lack of authorisation as inactivity, *e.g.*, to mask their reasoning process (+ $E_4$ ). Section 6.6.3.3 demonstrated how agents’ work went unimpeded by the unrelated (in)activity of their peers, *e.g.*, the processing of Amy’s workflow by St. Antonius was unaffected by SURF offering only an ignored task-execution plan (+ $E_5$ ). Unfortunately, to afford agents playing their BRANE roles, the runtime system had to store many statements, and the agents had to reason about their many complex interactions ( $-E_6$ ). Fortunately, this generalised several distinct processes of the original BRANE system, resulting in an implementation that is more concise by re-using this general solution (+ $E_6$ ). Moreover, as the JustAct framework is based on concepts that stakeholders in BRANE are expected to understand, our hardcoded portion of BRANE is smaller and simpler, and thus easier to reason about and maintain (+ $E_6$ ). Hence, complex BRANE-specific notions (*e.g.*, workers, domains, task execution, and asset providers) can be understood via JustAct concepts, which are common and well-understood in computer science (*e.g.*, agents, messages, signatures, rules, and first-order logic) and in legal regulations (*e.g.*, auditing, powers, actions, agreements, qualification rules, and evidence) (+ $E_6$ ).

Our system confers the BRANE system with (N)ew desirable characteristics. Via the complex Slick rules expressible in statements, domain-local policy providers can choose to internalise facets of their policies into the statements themselves (+ $N_1$ ). For example, in Section 6.6.3.4, St. Antonius used this feature in two cases. In Case 1, it enabled the explicit communication of a non-sensitive facet of the St. Antonius-local policy to the other agents. SURF benefitted from having more information to drive its local reasoning (+ $E_6$ ), and moreover, it effectively delegated the power to authorise some tasks from St. Antonius to SURF (+ $N_1$ ). By decoupling St. Antonius from some of its work, the system became more fault-tolerant, as new St. Antonius-authorisations could be created even after the St. Antonius agent or network crashed (+ $E_5$ ). In Case 2, St. Antonius internalised a sensitive facet of its local policy (+ $N_1$ ). The JustAct framework enabled just the agents trusted by St. Antonius to observe this information to communicate it, however, this control was external, and was not enforced using the usual action-justification mechanism ( $-E_3$ ). In either case, this delegated reasoning work from domains to other agents, potentially complicating their roles ( $-E_6$ ). But fortunately, agents retained the power to prefer inactivity, *e.g.*, to forget and ignore messages, to protect themselves from being overworked (+ $E_5$ ).

Throughout the scenarios, external effects at the domain-level (*e.g.*, the reading and writing of assets) and at the policy-level (*e.g.*, the proof that actions were justified) are connected via explicitly created, signed, communicated, and collected policy meta-data. On the one hand, the correspondence between user input and runtime behaviour is systematically driven, enforced, logged, and explained to the users concerned with correctness ( $+N_2$ ) and to the agents responsible for the actions ( $+N_4$ ). On the other hand, these activities are programmable via manipulations of the user inputs ( $+N_5$ ). Section 6.6.3.5 demonstrated that these manipulations can include the fundamentals of the inter-agent power dynamics which were hard-coded in the original BRANE, such as arbitrarily re-defining the notion of *involvement* that relates domain-local policies to the tasks whose execution they regulate ( $+N_5$ ). Fortunately, via the composition control features of Slick [EvB24], agents can express these changes by stating adjustments to policies, often without needing to synchronise with their peers. For example, in Section 6.6.3.2, Statement `bob 1` enforces the obligation to plan Bob’s four workflow tasks together before any are executable, and in Section 6.6.3.5, Statement `st-antonius 6` lets St. Antonius opt out of involvement in a particular asset, after judging that its contents are not sensitive information, creating new ways for workers to justify their processing of this asset ( $+N_5$ ). Throughout, agents preserved their autonomy over locally accessible data ( $+E_6$ ), *e.g.*, ‘Defector’ Dan could always express any data-access request, which were even realised without permission, but only if the system was configured to do so. But regardless, any actions without permission were systematically logged, to be detected by any auditors after the fact ( $+N_3$ ).

In summary, our system exhibited all the desirable characteristics to some extent. These improvements (and their limitations) resulted from the more unified and dynamic approach JustAct provides to regulating actions with policies. Our version of BRANE moves some of the burden from the static BRANE implementation to the agents at runtime, thus enabling greater flexibility than in the original BRANE, but requiring care to shield agents from the potential complexity of their interacting policies. The case study also exposed cases where the dynamism of JustAct still does not yet go far enough; *e.g.*, Section 6.6.3.4 demonstrates a scenario in which it would be useful if agents could use statements to regulate how agents communicate statements. In all scenarios, our version of BRANE benefits from its greater emphasis on accountability; more than ever before, it is clearer how to define, implement, and audit the policy-compliant execution of users’ medical workflows. This functionality was achieved with acceptable runtime performance.

These findings motivate further experiments in future to integrate the features of our experimental runtime system into the real BRANE system, and trying to reproduce these findings. To this end, work remains to evaluate the necessary facets of the real

BRANE system which are absent in these experiments: the physical distribution of agents and (meta) data, and the more extensively automated reasoning of agents that are automated in BRANE, including planners and workers.

## 6.7 Discussion of the Framework

This section evaluates the framework in general by reflecting on the processes and results of the implementations and experiments from Sections 6.4 to 6.6.

### 6.7.1 Strengths of the Framework

**Dynamic and Extensible** The framework is abstract, making it widely applicable.

Firstly, the framework is parametric to the statically-defined policy language, admitting any formal language affording pure functions of truth and validity. This leaves significant room for systems instantiating the framework to adopt various notions of policy, with various information models, and various semantics. For example, the framework's notion of policy affords the  $n$ -ary relations and logical constraints underlying Bell-LaPadua security policies and various Rule-Based Access Control policies, which are all summarised and compared in [ZC08], but we leave the development of such to future work.

Secondly, the relations between agents that ultimately control actions are highly dynamically configurable by agents making statements. This lets a wide variety of multi-agent runtime systems implement the framework. Different configurations confer different characteristics on the system, for example, allowing for dynamic specialisation for various use cases in reaction to runtime information. We recognise two noteworthy spectra on which particular system configurations fall. Together, these help to clarify the ways systems can change their characteristics at runtime. Firstly, systems can be centralised (where inter-agent consistency is high, *e.g.*, by most statements being agreements) or decentralised (where many agents can act independently). Secondly, systems can be highly static (where strict agreements are ex-ante enforced, resulting in predictability and efficiency) or highly dynamic (where actions are loosely specified and enforced, resulting in flexibility and autonomy).

**Formal Inter-Agent Power Dynamics** The framework is useful if agents agree to preserve their well-behavedness, as it is defined in Definition 6.1: agents take only permitted actions. Thus, policies with complex conditions for permission create complex inter-agent power dynamics, for example, modelling various common and useful normative concepts. For example, in Section 6.6.3.4, by making Statement `st-antoni` 5, the St. Antonius hospital formalises their trusted peers in the `x trusts y` relation, and also delegates their own power of authorisation to their

trusted peers. Despite its complexity, the policy expressed by Statement `st-antoni` 5 has unambiguous meaning. Agents will necessarily agree in which cases this statement contributes to a valid permission of a given action.

Well-behavedness is also robust; an agent that misbehaves by taking non-permitted actions preserve the well-behavedness of their peers. This affords meaningful cooperation between well-behaved peers even in the presence of misbehaving peers. The framework lays the foundation for agents to recognise and punish these bad actors, for example, by cutting them out of future cooperations, ignoring their statements, or affording their punishment by participating in future audits of their actions.

**Autonomy and Parallelism** Agents synchronise to change agreements. All other communication can be asynchronous, delayed, and lossy. Agents are very autonomous, as they are never fundamentally compelled to act or make statements. Hence, agents are robust to unreliable peers. The framework affords realistic inter-agent enforcement of well-behavedness: agents monitor actions, and actors bear the burden of proving that their actions are permitted. Thus comes at the cost of burdening actors: they are responsible for justifying their own actions. However, in cases where this burden is too great, agents are free to remain inactive instead; inactivity does not require justification. The framework is also robust to agents arbitrarily forgetting (their local) statement and action information; when these are not part of their existing justifications, this even preserves their well-behavedness.

**Consistent Permission despite Privacy** Agents always agree which actions are permitted, despite being defined by statements not known to all agents. This apparent contradiction is resolved by agents being able to decide when their known statements suffice for permission. Moreover, permission is objective; agents are certain that other agents (*e.g.*, future auditors) agree that their actions were permitted, without involving them at all. For example, in Section 6.6.3.4, the SURF agent can justify actions using Statement `st-antoni` 6, despite this statement being kept private between SURF and St. Antonius. This sensitive statement must only be observed at the moment the permission of this action is audited. For example, it can be audited by a third party trusted by St. Antonius not to reveal Statement `st-antoni` 6, and trusted by the other agents to accurately judge the permission of the action.

## 6.7.2 Limitations of the Framework

**Costly Justification Search** To remain well-behaved, agents cannot act until they have found a justification, by searching through combinations of known statements. In general, this problem is hard, scaling exponentially with the number of statements (the cost of enumerating combinations) and whatever is the language-specific cost of

evaluating large policies (the cost of evaluating composite policies). But in practice, the difficulty is curbed by limiting the consideration of statements to a smaller set of simpler policies. Fortunately, because agents act at their own pace (or not at all), each agent can approach this problem as best suits their needs, *e.g.*, by ignoring any statement that they expect to be more complex than it is worth. The cost of justification search can also be curbed more systematically by limiting the expressivity of the policy language. For example, in our case study, we found Slick to be sufficiently complex to express the necessary concepts, yet its policies were evaluated at a speed that was sufficient for our purposes (see Section 6.6.4), despite our implementation not being optimised for performance.

Future work can continue the design of policy languages and implementations and their effect on the cost of justification search. We are also interested in leveraging existing literature and tools to let agents more automatically and efficiently search for justifications and reason about their properties. We note two particularly promising approaches: 1. answer-set solving, *e.g.*, with Clingo [GKK<sup>+</sup>11], and 2. rewrite systems modulo theories, *e.g.*, with Maude [CDE<sup>+</sup>03]. For the former, we intend to draw from work in Chapter 2: we formulate the search for facts to state as search problems and let the Clingo reasoner search for solutions.

Alternatively, we consider turning to generative AI to suggest which policies should be stated in order to reach goals expressed in prompts, for example, beginning with ‘Please make a Slick statement that completes the following justification’. We are inspired by work such as CoqPilot [KSKP24], which exploits the creativity of generative AI to write machine-checkable specifications and mitigating the unreliability of the generation process by filtering the output via conventional correctness checks; in our context, the Slick interpreter can easily check which generated policies contain are well-formed, achieve the desired result, and preserve validity.

**No Obligation to Act (in Time)** Our framework offers no fundamental mechanism for agents to compel one another to action. Thus, the framework cannot faithfully internalise normative *obligations to act*. Agents can approximate obligations by conditioning future permissions on proof of their prior actions. This approximation was sufficient to implement the functionality of BRANE in Section 6.6. Also, any required enforcement of obligations can be built atop the framework, implemented externally, using supplementary systems. However, we expect that agents may have difficulty compelling agents to act before specific deadlines, as a consequence of the aforementioned point on the costly search for justifications. For this reason, our framework is not a natural choice for implementations of *real-time systems*, where agents must react to stimuli within strict deadlines. Future work can investigate

constraints on system configurations that strike desirable compromises between system flexibility on the one hand, and predictability of actions on the other.

**Specification of Communication** It is a key feature of the JustAct framework that permission of given actions is universally agreed by agents, despite each having only a partial view of the existing statements and actions; *i.e.*, the framework works despite agents gossiping statements and actions at their own pace. However, as discussed in Section 6.6.4, and exemplified in Section 6.6.3.4, the permission mechanism does not extend to specifying this gossip; *i.e.*, the framework does not afford agents using statements to specify how statements and agreements are communicated. In future work, we want to supplement the current notion of agreements and permissions with a similar mechanism for the communication of statement and action information, which regulates the sending of statements and actions from one agent to another, and is violated when the information is sent without permission.

**No Privacy from the Actor** Agents are only able to act on permissions defined by *known* statements. For example, in Section 6.6.3.4, for SURF to act while remaining well-behaved, it is necessary for SURF to acquire the sensitive statement `st-antoni` 6 from the St. Antonius hospital. Moreover, as discussed in Section 6.7.1, auditors of the permission of this action must also observe the statement. At present, these cases can be worked around by hiding the reasoning behind the permission. For example, in this case, some agent can be empowered to decide whether the St. Antonius trusts another given agent, and can hide the reasoning process dependent on the sensitive information previously expressed in statement `Statement st-antoni` 6. The information can be preserved by (privately) logging a systematic transformation of the reasoning in its entirety to a restricted version, which can be publicised, because it omits the sensitive details. This kind of information hiding has plenty of precedent in the literature. It is even commonplace in the current version of the BRANE system: checkers publicise authorisations, but hide any underlying reasoning. This workaround has the drawback of hiding the reasoning in undesirable cases, *e.g.*, it is hidden from trusted auditors. In future, we want to systematise these intermediary transformations such that the hidden policy information can always be recovered.

## 6.8 Related work

Our framework prescribes a relation between concerns that are each independently explored in the literature: 1. blockchains can synchronise dynamic, decentralised policies, 2. trust management specifies the delegation of power between agents, and 3. Curie evaluates data access policies as a function of sensitive data.

### 6.8.1 Smart (Policy) Ledgers atop Blockchains

Distributed ledgers are an abstraction of consistent state over decentralised processes. Blockchains afford robust, probabilistic implementation of distributed ledgers, but they differ in their details. For example, Fabric emphasises scalability [ABB<sup>+</sup>18], while Ouroboros emphasises provable security [KRDO17].

SmartAccess [dORV<sup>+</sup>22] uses distributed ledgers to store policies and (meta-)data, enabling decentralised implementations of the access-control model whilst affording accountability regarding authorisation. However, this approach imposes a burden on agents to maintain a high level of synchrony in the policy information they create and collect. This imposes unnecessary overhead in cases where policy information is unrelated. For example, the concurrency demonstrated in Section 6.6.3.3 would be impossible; agents would be forced to synchronise their statements just in case they affect the permissibility of their actions. Moreover, the synchronisation of policies would make it impossible to support cases where policy information captures private information, and thus, should not be broadly publicised. This is demonstrated in Section 6.6.3.4, where the St. Antonius hospital sends the (trusted) SURF agent some private information to enable new permitted actions.

Other ledger-based systems allow a heterogeneous view on the policy state. For example, Canton [Can24] (whitepaper) replaces the (sequential) *blockchain* with a (hierarchical) *blocktree*. Agents must only synchronise the relevant sub-trees with their neighbours. This lays the groundwork for private policies. This idea is promising and requires further investigation.

We see the various, established ledger and blockchain technologies as ways for our agents to synchronise agreements in particular. But they are unsuitable for distributing all policy information, where ordering and synchronisation is often unnecessary or undesirable.

### 6.8.2 Curie

Curie is a policy-based data exchange system [CAA<sup>+</sup>19]. Our works share a fundamentally decentralised approach to the specification and enforcement of formal policies that regulate the exchange of data, based on consortium agreements and local policies. Moreover, both works define permission to act in terms of shared, composite policies. In the case of Curie, data sharing is composed from the policies of agents  $x$  and  $y$ , requiring their dual views on the sharing:  $x$  must specify that data is shared with  $y$ , and  $y$  must specify that data is acquired from  $x$ . We also see similarities between Curie and the EPI framework in their shared application to federated machine learning with sensitive medical data.

The contributions of Curie and JustAct are largely orthogonal, because they are designed to address different kinds of complexity in policies. Curie focuses on complexity arising from the definition, sharing, and regulation of policies following their entanglement with the shared data itself. Policy decisions digest the homomorphically encrypted data [DMGD23], revealing only the evaluation result. For example, a hospital’s sharing policy is conditioned on the data surpassing a threshold of differential privacy [MZ17]. In contrast, JustAct focuses on multiple agents cooperating to define and disseminate policy information on the fly, while preserving the accountability and autonomy of the actors under regulation.

Because Curie and JustAct make largely orthogonal contributions, we see great potential in investigating the combination of their best features. Can we develop runtime systems in which data sharing actions are regulated by policies defined by cooperating agents, as a function of the data itself, while preserving the agents’ accountability, privacy, and autonomy? Can (data sharing) actions be justified as a function of the shared data itself? Can the Curie language be used to express policies to the satisfaction of the JustAct framework requirements? In which applications is this combination fruitful?

### 6.8.3 Trust Management

Traditional access control develops languages and tools for specifying and checking a requester’s permission to access data. *Trust management* reifies the role of the accessor as a *certificate*, primarily, to enable access control in a decentralised environment, where the identities of particular requesters are not known ahead of time [BFL96]. Much literature dates to the 1980’s and 1990’s, investigating policy languages suited to defining certificates and inferring them at request time from context. Many of these are extensions of Datalog, *e.g.*, adding non-monotonicity [LGF03], constraints [LM03], and weights [BMS08].

Like access control, trust management focuses agent reasoning on the access-request decision, whereas our framework emphasises the inter-relationship between agents and their actions via their synchronised agreements. However, the bulk of trust management research complements our work, because it informs the selection of particular policy languages suited to particular purposes. [Sac11] overviews and compares (the complexity of) noteworthy trust management languages. Which trust management languages are good candidates for policy languages in the JustAct framework? We expect these to be even more desirable if the framework is extended to support the desirable features explained in Section 6.7.2, empowering policies to specify how agents can communicate statements and actions.

## 6.9 Conclusion

In this chapter, we define *JustAct*, a framework for systems driven by autonomous agents whose actions on the system are regulated by the policy information that they define, communicate, and assemble. Our work is motivated by the problem of exchanging sensitive (*e.g.*, medical) data between autonomous agents, where it is crucial that 1. agents can define adjustable policies that regulate their peers' usage of their data, and 2. agents can decide to withhold the policy information that justify their actions, because the policies themselves sometimes capture sensitive information.

Our framework is instantiated by runtime systems whose behaviour is unfolded at runtime by multiple, autonomous agents communicating statements (carrying policy information) and taking actions (whose effects connect to events that are external to the regulated system). To maximise the applicability of our work to existing literature, future implementations, and in existing systems, we do not fix the language agents use to express policies. In general, it suffices for the language to meet our minimal specification. Intuitively, each policy must be communicable in messages, and determine the *validity* and *effects* of a given action. Crucially, agents agree whether a given action is *justified* by a given policy at a given time. Hence, actors can decide and prove that their actions are permitted, such that future auditors come to the same decision. The space of justifications remains flexible, yet unambiguous, because agents maintain consensus on the changing set of *agreements* which are the basis of future actions.

We evaluated our framework in application to a representative case; we apply a Rust implementation of JustAct in a runtime system and policy interpreter to usage scenarios of BRANE, an existing medical workflow execution system. Desirable features are reproduced successfully: data scientist users can define workflow tasks, workers can execute them, while domain-checkers regulate the processing of domain-controlled medical data. Users enjoy the typical benefits of BRANE: domain-policies are programmable, and data-independent tasks complete concurrently. Moreover, our approach affords greater expressivity and flexibility in the regulation of workflow execution. For example, we demonstrate agents dynamically delegating their power to authorise workflow tasks, and amending the roles of agents in workflow execution, while preserving their consensus on which actions (in the past and future) are permitted, and what are these actions' effects on the medical data.

We discussed the generalisability of these findings to other systems, remarking on the strengths and limitations of our approach. The strengths follow from systematising the connection between agents and programmable policies, laying the groundwork for unified approaches to various tasks that are difficult when they are separated; for

example, our agents use policies to 1. formalise the connection between system events and external sources of regulatory norms such as the GDPR, 2. audit the permission of existing actions, 3. plan actions that have desirable effects, and 4. identify policy amendments and changes that afford desirable actions.

There are two kinds of limitation to our work. Firstly, the additional flexibility and dynamism our framework demands of the system comes at the cost of computational complexity: agents striving to act must search for the missing statements, possibly coercing them from their peers. To some extent, this is unavoidable, arising as a necessity of systems being designed to exploit the novel flexibility we provide to the dynamics between agents. However, the logical, systematic nature of our policies affords the application of existing language-design and system-analysis techniques to designing policy languages and policies to strike better compromises between flexibility and simplicity. Secondly, we have identified extensions of this framework that push it yet further in the direction of linking policies to agent behaviour. For example, we see promise in future work extending the notion of permission; we want statements to regulate how statements are communicated.

**General Takeaways** JustAct adapts our notion of cooperative specification to a context where each agent decides if and how to share parts of the specification. The key idea is that it suffices for agents to agree on *how to decide* if a given action is permitted. We let this be defined by the *agreement*, the only part of the specification that the agents keep perfectly synchronised, and which standardises their process of auditing (the permission of) any action. The burden is on actors to present sufficient policy information to justify their own actions on a case-by-case and need-to-know basis. We demonstrate how JustAct is applied in the medical domain, where medical metadata cannot always be shared, in order to protect the privacy of data subjects. Here, the framework clarifies how data processing policies are amended and added piecemeal, while clarifying how agents' data processing is reliably audited.

## General Conclusion

### 7.1 Reflection on the Thesis Contributions

**An Overview of the Findings** Chapters 2 to 6 explored various specifications of (data exchange) systems, and how these specifications could be used.

Chapter 2 showed how eFLINT specifications – which are formulated to mirror legal concepts – could be reliably translated into a more abstract representation, based on first-order logic – which connects to many other domains, and whose theory and tooling are well-developed. We exploited this theory to address a weakness in eFLINT’s existing semantics. Then we exploited this tooling to enable a powerful new usage of the usual eFLINT specifications: users characterise eFLINT scenarios of interest, and leave the difficult search process to automated tooling.

Chapter 3 showed how these same logical foundations could capture stakeholder requirements in a new context, where agents cooperate to incrementally specify their requirements on the system. The technical novelty was reusing the usual specification rules for controlling how agents cooperate. Thus, a precisely defined language leads to precise control over the cooperation. We showed how this framework was sufficiently general to capture and combine various requirements on data exchange systems: legal norms, archetypes, workflows, software events, and so on.

Chapter 4 generalised the framing of this cooperative specification problem to a whole class of specification languages. We experimented with various combinations of existing languages, control mechanisms, and usage scenarios. We found that different languages resolved the fundamental tensions between different ideal language qualities by making different trade-offs. We found cases where each language had its own interesting applications. We proposed SEASO and Slick as two new cooperative specification languages, that make new trade-offs between existing language features.

Chapter 5 showed that the incremental, cooperative specification of the system could be interleaved with its automated enforcement, even when agents are distributed over a (physical) network. In this case, specifications are *protocols*, specifying the

messages in flight over time, in a bespoke protocol language: PDL. To make this possible, we co-designed PDL with the distributed runtime system, such that the protocols and the runtime have key properties, which we formalised and verified with Coq. Firstly, the PDL semantics is *(de)compositional*: we proved that composing PDL protocols intersects accepted behaviours. This made it possible for agents to reason about behaviour in ignorance of their peers' protocols. Secondly, the runtime system was restricted to *constructive* behaviours, which are accepted by definition, but which are computable from the protocol itself. This made the runtime implementation feasible, because it let the protocols 'guide' the system behaviour at runtime.

Chapter 6 defined the *JustAct* framework, which combines the language-agnostic cooperative specification framework of Chapter 4 with the coordinated distributed system of Chapter 5. Here, specifications are *policies*, regulating agents' actions. The novelty was in how agents create, communicate, and reason about these specifications. The framework systematises the balance between policy consistency on the one hand, and agent autonomy on the other, as a function of which policies the agents agree to synchronise. More permissive agreements let agents work more independently, because they depend less on their peers. We show how our framework ensures flexibility and reliability at the same time: inter-agent power dynamics are changed on the fly, whenever agents synchronise new agreements. But nevertheless, agents can always decide when they have collected enough evidence that an action is permitted, and they can prove it to anyone forever, for example, to future auditors. We showed how our prototype data exchange system recreated BRANE, an existing medical workflow processing system, but our framework improved its flexibility and reliability. For example, at runtime, we changed BRANE's (previously hardcoded) *involvement* relation between agents, workflow tasks, and datasets.

**Specification-Centrism** The contributions of this thesis explored many applications of the same idea: once you have a semantic *model* of the behaviour of your system, many activities are unified in the manipulation of *specifications* of that model. Through this lens, for example, Chapter 3 proposed that changing the specification *is* system behaviour that should be specified. And Chapter 5 proposed that the specification language *is* the network endpoint API.

Specification-centrism is promising, because it relates many important activities. For example, because Chapter 2 lets Clingo programs model-check eFLINT norms and Chapter 6 lets Slick policies control agent actions, relating eFLINT and Slick specifications lays the groundwork for model-checking of agents' actions. In general, the more thoroughly we connect these activities, the more meaningful each specification becomes, and the more facets of our systems we can design, predict, negotiate, and control via the same typical formal methods, yielding the typical benefits.

**Contributions to Data Exchange** The later chapters showed how several facets of data exchange systems can be controlled and automated. Here, the specifications played a vital role in making these systems practical. Firstly, they gave (human) users intuitive and appropriate abstractions over data exchange systems, which otherwise would be overwhelming in their complexity and rate of change. Secondly, the precise semantics of specifications acted as contracts between users, such that their disputes are resolvable, and between users and automated services, such that humans retain control despite extensive automation. Thirdly, by centring systems on the enforcement of specifications, flexibility becomes ‘programming’, which is an area where we embrace change, perhaps in contrast to ‘system architecture’. I hope that any data exchange systems deployed in reality have these qualities, so that we understand how they work, we stay in control of what they do, and we do not waste resources replacing them much more than we have to.

The early chapters showed how several requirements on data exchange systems could be expressed via formal specifications. As usual, the results of this process are useful, because they lay the groundwork for systematic analysis and verification. But the process itself is also useful, because it exposes ambiguities and assumptions that might have gone unnoticed until they caused serious harm. For this reason, I hope that my work will convince newcomers to consider using some formal languages and tools for their expression and reasoning. I believe that Lamport said it best.

*A less obvious reason to improve your writing is to improve your thinking. [...] If you think you understand something and don't write down your ideas, you only think you're thinking. – dr. Leslie Lamport, on the 5 minute mentor podcast.*

## 7.2 Avenues for Future Work

Despite my best efforts, lines of research exploded in several directions faster than I was able to bring them together. As a result, you may have noticed that each contribution chapter has its own long list of directions for future work. Here, I briefly highlight the most promising ones.

- In some form, every contribution chapter explored the complex relationship between ‘hard’ technical properties of a given specification language and the ‘soft’ usability qualities that emerged in certain application contexts. I consider it important to continue investigating variations of these languages which improve the **explainability** of semantic objects, because I believe this is one of the most significant blockers to the adoption of these languages in practice. For example, we discussed how our new semantics for eFLINT has many desirable qualities (*e.g.*, enforced properties are easy to express via rules), but it still has shortcomings.

Notably, subtle contradictions are still too easy to express, and they impede explainability too significantly. We have explored ways to improve explainability (*e.g.*, by switching to the well-founded semantics), but they require sacrificing other qualities. Are there better semantic foundations that will let these qualities coexist? Are there better ways for us to think about explainability?

- Now that we have explored which facets of policies we can connect to notions of ‘permission’ of systemic actions (in Chapter 6), can we **reconnect these policies to legal theories**. Recall that this question was the impetus behind FLINT and eFLINT in the first place (in Chapter 2). Are FLINT and eFLINT the best choices for formalising legal norms in this context? Or does the simplicity of Slick, which simplified our examples in Chapter 6, make complex policies easier for humans to reason about? Perhaps the ideal normative specification language has a mix of these features. More experimentation and case studies are required.
- Now that we have found ways to integrate specifications deeply into our runtime systems, it has become particularly appealing to **automate the search for desirable specifications**. Most immediately, we want to encode the search for Slick specifications (as in Chapter 6) as search problems in Clingo (as in Chapter 2). We have done some preliminary exploration, but we had no results that warranted inclusion in the thesis. Many different approaches are also conceivable, and may be superior. For example, are Maude’s rewrite rules a better fit than Clingo’s inference rules? What cases of ‘desirable’ can we express and support?
- Coq allowed us to precisely formulate and prove vital properties of Reowolf’s language and runtime system. We hope to apply this approach to **machine-verifying more aspects of our data exchange systems**, *e.g.*, particular systems instantiating our JustAct framework. The first steps are clear; we can encode the existing properties and requirements of the JustAct framework in Coq. But what comes next? Which requirements on these systems are really needed in practice, and for which of them are Coq proofs possible and necessary? Is it productive to try and bring these verification activities (*e.g.*, in Coq) closer to the specification-writing activities that our agents already perform (*e.g.*, in Slick)?
- When should **generative AI help users to write specifications**? We hope to benefit from the creativity of generative AI while mitigating its unpredictability with our usual symbolic reasoning. We imagine agents in Chapter 6 prompting AI tools to recommend policies that, when stated, would achieve new goals, *e.g.*, ‘write a Slick statement that justifies my processing of Bob’s X-Ray data’.

At the time of writing, it is my intention to do much of this future work myself. But I welcome others to join in or to beat me to it, if they haven’t already.

---

## Summary

Various industrial and research projects are underway, developing organisations, infrastructure, and software to support the safe sharing and processing of data across organisational boundaries. These support various use cases. For example, universities collect data from hospitals to study their records on rare medical diseases, and airline companies trade aircraft sensor data to optimise aircraft maintenance. Some projects aim to bring these efforts and their developments together by developing common procedural standards, sharing platforms, and software frameworks. In general, all these developments must enforce the accountability of the data consumers, to enable the enforcement of organisational and legal norms such as data protection regulations. And moreover, the developments must remain flexible to changes in the norms, so that the systems stay under control as laws are amended or as users come and go.

As part of the work behind the AMdEX-Fieldlab and AMdEX-DMI projects, this thesis develops formalisms and tools for case-generic, safe, and flexible data exchange software systems. Several contributions are included; they have in common that they centre the control and reasoning about the system's behaviour around the incremental manipulation and analysis of shared *specifications* of how the system and its users are permitted to behave. The first contribution chapter improves the existing eFLINT specification language. This clarifies the connection between legal norms and logical models of system behaviour, and it opens the way for new forms of automated case analysis. The middle chapters develop and study new eFLINT-like languages which support the safe refinement of the shared specification by multiple agents with limited knowledge of one other. For example, a legal expert can formalise the legal notion of 'conditional consent to data processing', and can specify the ways in which the defined conditions may be changed by system administrators and data controllers, depending on their relation to the data in question. The last contribution chapters co-design protocol and policy specification languages with software frameworks and runtime systems, such that safety and flexibility are preserved when the users and their specifications are physically distributed over computer networks.

---

## Samenvatting

Verschillende industriële en onderzoeksprojecten zijn aan de gang met het ontwikkelen van organisaties, infrastructuur en software om data veilig te delen en te verwerken buiten de grenzen van een organisatie. Hiermee worden verscheidene casussen ondersteund. Zo verzamelen universiteiten data van ziekenhuizen om die te bestuderen voor het detecteren van zeldzame ziektes, en wisselen luchtvaartmaatschappijen onderling data van sensoren om vliegtuigonderhoud te optimaliseren. Bepaalde projecten hebben als doel om deze inspanningen en de bijbehorende uitkomsten te verenigen via de ontwikkeling van gemeenschappelijke procedurele standaarden, deelplatformen en software raamwerken. In het algemeen moeten al deze ontwikkelingen de verantwoordelijkheid van de gegevensverwerkers kunnen afdwingen, zodat naleving van organisatorische en wettelijke normen, zoals regelgeving voor gegevensbescherming, gewaarborgd blijft. Daarbij moeten de ontwikkelingen flexibel blijven om verandering van normen te ondersteunen, zodat de onderliggende systemen beheersbaar blijven wanneer wetten worden aangepast en wanneer gebruikers komen en gaan.

Als onderdeel van het werk binnen de AMdEX-Fieldlab en AMdEX-DMI projecten ontwikkelt dit proefschrift formalisering en hulpmiddelen voor de ontwikkeling van casusafhankelijke, veilig en flexibele software systemen die gericht zijn op het uitwisselen van gegevens. Verschillende bijdrages hebben gemeen dat de focus ligt op het controleren en redeneren van en over het gedrag van het systeem ten opzichte van de incrementele manipulatie en analyse van gedeelde *specificaties* die beschrijven hoe een systeem en de gebruikers zich dienen te gedragen. Het eerste bijdragende hoofdstuk verbetert de bestaande specificatie language genaamd eFLINT. Dit verheldert de connectie tussen juridische normen en logische modellen van systeemgedrag. Ook maakt het nieuwe vormen van geautomatiseerde casusanalyse mogelijk. De middelste hoofdstukken ontwikkelen en bestuderen nieuwe, op eFLINT geïnspireerde talen die het mogelijk maken dat meerdere partijen, met beperkte kennis van elkaar, veilig de gedeelde specificatie kunnen verfijnen. Zo kan een juridisch expert het concept van 'voorwaardelijke toestemming tot gegevensverwerking' formaliseren, en kan de expert specificeren hoe de condities aangepast kunnen worden door systeembeheerders en verwerkingsverantwoordelijke. De laatste bijdragende hoofdstukken ontwikkelen specificatietalen voor protocollen en beleidsregels samen met software raamwerken en uitvoersystemen, zodat de veiligheid en flexibiliteit behouden blijven wanneer gebruikers en hun specificaties fysiek verdeeld zijn over computernetwerken.

---

## Glossary and Abbreviations

AI Artificial intelligence refers to a wide range of research topics and technologies.

Historically, it characterised sophisticated symbolic reasoning engines, but more recently it is mostly used to refer to very large neural networks, *e.g.*, for generating natural language text from prompts. 1, 240

Alloy is a constraint- and relation-based specification language used to model and specify software systems and domain-specific languages [He06, Jac19, Jac03, MP15]. The AlloyAnalyzer tool model-checks Alloy specifications. See the website at <https://alloytools.org> for details. 7, 9, 11, 12, 60, 95, 99, 102, 111–117, 125–129, 131, 134, 137, 139–141, 229

AlloyAnalyzer is the standard analyser of Alloy specifications. Users direct its bounded search for instances satisfying the given Alloy specification. 112, 113, 115, 129, 229

AMdEX The Amsterdam Data Exchange refers to a sequence of research projects AMdEX-Fieldlab and AMdEX-DMI, which develop(ed) knowledge and tools for case-generic, safe, lawful, and inter-organisational exchange of data. AMdEX includes primary research (mostly at UvA) alongside efforts of standardisation, implementation, and application work with industrial partners such as KPMG. The AMdEX website is <https://amdex.eu>. Here, and in Chapter 6, AMdEX also refers to the consortium of the stakeholders and members in these projects. Sometimes, AMdEX refers to an organisation that stewards the adoption and maintenance of the AMdEX developments. iv, 3, 7, 12, 229, 232, 234, 237

AMdEX-DMI succeeds the AMdEX-Fieldlab project that began in 2024 (Dutch Metropolitan Innovations ecosystem for smart and sustainable cities, made possible by the Nationaal Groeifonds) and focused on the operationalisation and standardisation of data exchange technology within the DMI ecosystem. iii, 3, 58, 227–230

AMdEX-Fieldlab is an EU project which ran from 2021–2023 (Kansen Voor West EFRO grant KVVW00309 at <https://competition-cases.ec.europa.eu/cases/SA.62475>) that funded my PhD research. The project mixed fundamental research at the UvA with standardisation, implementation, and testing work alongside industrial partners such as KPMG. The project was succeeded by the AMdEX-DMI project. iii, 3, 58, 227–229, 232

API Application Programmer Interface is a generic term for any description of the interface between a programmer and an application. APIs may be function signatures, a formal language, constraints, or communication protocols. 143, 145–147, 161, 166, 224, 230, 231, 234

BRANE is a modular and distributed workflow processing system, implemented in Rust, that separates the concerns of specialised users: software developers provide containerised functions, data scientists define workflows, and system administrators maintain the worker processes [VCB21]. Originally, BRANE was styled ‘Brane’ and it abbreviated nothing [VCB21]. BRANE has been reappropriated as main component of the EPI framework, which focuses on medical workflow processing. 9, 14, 167–170, 195–202, 206, 208, 211–215, 217, 218, 221, 224, 236

BSD The Berkeley Software Distribution refers to an operating system (and its variants) developed in Berkeley. Its socket API was widely adopted, and influenced the Reowolf project. 9, 12, 13, 145, 166, 231, 236

BSP Bulk Synchronous Parallel is a processing model for distributed computer systems [CFSV95]. Work is spread over linear *supersteps*, within which, each process works on local data, and between which, processes send and receive messages. 164

CCI Complex Cyber Infrastructures is a research group, previously led by prof. dr. ir. C.T.A.M. (Cees) de Laat and currently led by prof. dr. Sander Klous, at the Informatics Institute of the UvA. I am a group member, along with frequent co-authors Tim Müller and Thomas van Binsbergen. iv, 234, 238

Clasp is an answer-set solver of grounded logic programs [GKS12]. Together, Clasp and Gringo form the Clingo solver. 52, 233

Clingo refers to an answer-set logic programming language and its solver tool, and is a major part of Potassco. Clingo is used to model complex search problems; search steps and constraints coincide in Clingo rules. Clingo is used extensively

in Chapter 2, but it remains relevant in the later chapters also. iv, 8–10, 17, 21, 22, 24–32, 35, 36, 38–42, 46, 47, 49, 51–56, 71, 89, 106, 108, 126, 132, 137, 139, 173, 217, 224, 226, 230, 231, 233, 235, 240

connector is the API of the Connector Runtime that is similar to the BSD socket API, but it emphasises configurable multi-party sessions by being oriented around PDL protocols. 9, 12, 13, 143, 145–147, 165, 166, 231, 236

Connector Runtime is the distributed runtime system interfaced by connectors. The Connector Runtime unfolds rounds of communication behaviour by executing the session protocol, expressed in PDL. 7, 13, 14, 143, 145, 146, 160–162, 164, 166, 231, 235, 236

Coq is a dependently-typed functional specification language and proof assistant developed at Inria. Coq is used to model and reason about systems, *e.g.*, which model software systems. Coq formalises the definitions and proofs in Chapter 5. The Coq has very recently been renamed to Rocq. 13, 135, 137, 143, 145, 146, 224, 226, 233, 236, 246–253

Core eFLINT is an EDSL in Clingo developed in Chapter 2 to serve as a core language for eFLINT and an intermediate representation in our translation from concrete eFLINT. 21, 22, 30–32, 38, 39, 41, 51, 52, 54–56

CWI The *Centrum Wiskunde & Informatica* (English: Center (of) Mathematics & Informatics) is a Dutch research institute based in Amsterdam. I worked there as a scientific programmer in 2019–2020, working on the Reo compiler and the first phase of the Reowolf project. iii, iv, 8, 13, 233, 236

Datalog is a simple logic programming language that has changed little since it was first presented decades ago. Since then, there have been many developments of variants of the language and tools (of these variants) for all sorts of modelling and reasoning problems. [CGT89] gives a good introduction of the state of the art in 1989. [MTKW18] gives a more modern overview. 7, 9, 10, 12, 24, 26, 53, 60, 62, 65, 71, 88, 92–94, 99, 102, 105–110, 114, 115, 117, 123, 125–134, 137, 139–141, 169, 172, 173, 220, 231, 237

Datalog<sup>¬</sup> is one name for the common extension of Datalog to include *weak negation*: conditions may be negated with ( $\neg$ ), and each  $\neg x$  holds if and only if  $x$  is false, *i.e.*, *not* inferrable from program rules. 169, 172–176, 190–193

DIPG Diffuse Intrinsic Pontine Glioma is a rare disease, a kind of brain tumour in children. Its study motivated the development of a large medical data consortium [BBL<sup>+</sup>17]. 1, 2

DMI The Dutch Metropolitan Innovations Ecosystem consists of various projects and organisations concerned with social and infrastructural projects in urban Dutch environments. DMI became affiliated with AMdEX in 2021. 43, 229, 232

DSL A domain-specific language is a (formal) language designed for the needs of a particular application domain. For example, eFLINT is a DSL specific to the domain of cyber-social norms, which is reflected in its abstractions of power, duty, action, obligation, and so on. 18, 111, 232

EDSL An embedded DSL is a language that is encoded entirely within another host language. EDSLs are implemented as software libraries, but they are used as DSLs: they define a new semantic abstraction. We define Core eFLINT as an EDSL in Chapter 2. 29, 231, 232, 234, 235, 238

eFLINT Executable FLINT is a domain-specific language for modelling and specifying norms in cyber-social systems. It was developed for case analysis of FLINT specifications in the SSPDDP project, and then presented in [vBLvDvE20]. eFLINT is the subject of Chapter 2. 6–12, 17–24, 26, 27, 29–36, 38–47, 49, 52–56, 59, 60, 64, 79, 87, 93, 94, 99–102, 111, 116–132, 134–141, 172, 191, 223–228, 231–234, 236, 237, 239–241

EPI the Enabling Personalised Intervention project, whose homepage is <https://enablingpersonalizedinterventions.nl>, studied the safe sharing and processing of medical data between organisations. Members of EPI and EPI overlapped and collaborated extensively. iii, 169, 195, 197, 219, 230, 232, 237

Esterel is a synchronous protocol specification language designed for real-time reactive computer systems. Research on Esterel emphasises static composition of imperative, message-passing programs, such that the resulting program’s runtime behaviour is efficient and predictable. Esterel influenced our formalisation of PDL in Chapter 5. 9, 143, 144, 163

EU The European Union is a political entity comprised of European countries. The EU or the Netherlands in particular funded most research projects relevant to this thesis, including AMdEX-Fieldlab, Reowolf, and SSPDDP. Consequently, many of these projects’ developments are guided by EU law; *e.g.*, eFLINT has seen extensive application to the EU GDPR. 1–3, 58, 100, 171, 230, 232, 233, 235

FIEVeL Functions for Institutionalized Environments Verification Language is a language for specifying the policies of multi-party institutions, affording their model-checking against properties via translation to PROMELA. 9, 94, 172

FLINT is a knowledge representation language for social norms, developed first at the CWI, and later (jointly) at the UvA law school and informatics institute. In [vDvdSvE16], FLINT abbreviated Formal Language for the Interpretation of Normative Theories, but recent works on FLINT and eFLINT like [vBLvDvE20] do not expand the abbreviation.. 7–9, 226, 232

FOPL First-Orde Predicate Logic is a formal system or language, attributing semantic meaning to syntactic formulae built from logical variables (*e.g.*,  $X$  and  $Y$ ), propositional connectives (*e.g.*,  $\rightarrow$  and  $\vee$ ), and quantification and checking of members of predicates (*e.g.*,  $\forall x, P(x) \rightarrow Q(x)$ ). 84–86

GDPR The EU General Data Protection Regulation is an EU law that governs how personal data of individuals in the EU is collected, used, and protected [Eur16]. Much work relevant to this thesis formalises and enforces these kinds of laws, so GDPR articles are often used in examples. 58, 59, 63, 64, 100, 111, 171, 222, 232

Gringo is a grounder of logic programs with variables, *i.e.*, generating equivalent variable-free rules [GHK<sup>+</sup>15, HLY13]. Together, Clasp and Gringo form the Clingo solver. 52, 230

Haskell a statically-typed, lazily-evaluated, and pure functional programming language. Haskell is the implementation language of (both versions of) the eFLINT interpreter. 11, 18, 30, 40, 41, 135, 136, 165, 248, 249

Inria The French Institute for Research in Computer Science and Automation (French: *Institut national de recherche en sciences et technologies du numérique*) develops Coq, which enabled our work in Chapter 5, and hosts several of my colleagues; notably, dr. Benjamin Lion is my co-author of [ELHA25], and prof. dr. Benoit Combemale is in my thesis opposition committee. 231

IoT The Internet of Things refers to research into small and specialised networked devices [REC<sup>+</sup>15], *e.g.*, sensors and smart kitchen appliances. 165, 171

IP Internet Protocol is a packet-switching protocol in the network layer of the OSI stack, where messages are routed and (de)fragmented. IP remains integral to computer networks. 146, 161, 170, 233

IPv4 Version 4 of IP, which remains prevalent since its creation in the 1980's. 146

ISO The International Organisation for Standardisation is responsible for many notable international standards, such as the OSI model of layered computer network communication systems. 235

Java is a memory-managed, object-oriented, and general-purpose programming language, which is traditionally compiled to Java Bytecode, which is executed by the JVM. Java remains popular for large-scale and industrial applications. 11, 12, 102, 135, 234

JSON Javascript Object Notation is a textual data format of (nested) arrays and lists. JSON is widely adopted for (de)serialisation of data for network transport or simple EDSLs. For example, the network API of the server wrapping the eFLINT interpreter is embedded in JSON. 235

JustAct is a multi-agent runtime system framework ‘where agents justify their actions’. Its is the subject of Chapter 6. 9, 14, 16, 167, 173, 177, 180–183, 185, 187, 188, 190, 192, 195, 209, 213, 214, 218, 220–222, 224, 226, 236

JVM the Java Virtual Machine is an interpreter of Java bytecode. The JVM provides a platform-agnostic abstraction and compilation target for Java compilers. 234

KPMG is a multi-national financial accountancy and consultancy firm originally named after its founders (Klynveld, Peat, Marwick, and Goerdeler). KPMG participates in AMdEX, *e.g.*, via prof. dr. Sander Klous, the group leader of CCI. 229, 230

Maude is a specification language and toolchain based on a rewriting logic. See the article [CDE<sup>+</sup>03] and the webpage [https://maude.cs.illinois.edu/wiki/The\\_Maude\\_System](https://maude.cs.illinois.edu/wiki/The_Maude_System) for more details. Maude is used for modular and specification and model-checking of complex dynamic (software) systems. 89, 226

MPI The Message Passing Interface is an open API standard for multi-process computer systems which is maintained by the MPI forum (<https://www.mpi-forum.org>). The MPI standard prescribes (abstract) syntax and (partial) semantics of MPI implementations, which exist and are used in various languages such as Java and Python. 164, 165, 234

MPST Multi-Party Session Types are a formal framework for specifying multi-party, stateful, communication protocols. MPSTs are used to formalise, verify, and check the safety of communication code, *e.g.*, in Java programs. 165

- NGI The Next Generation Internet is a part of the EU's Horizon Europe research initiative running from 2021 to 2027. NGI funded the Reowolf project which is relevant to Chapter 5. 236
- nuXmv The New ( $\nu$  'nu') Extended Symbolic Model Verifier refers to a finite-state symbolic model checker tool and its input language. 55, 89
- ODRL The Open Digital Rights Language is an EDSL in XML or JSON used to specify policies describing the permissions for use of (digital) resources. The ODRL information model is available at [www.w3.org/TR/odr1-model](http://www.w3.org/TR/odr1-model). 64, 171
- OSI The Open Systems Interconnection is an ISO model that organises computer-networked system communications into seven distinct layers: 1. physical, 2. link, 3. network, 4. transport, 5. session, 6. presentation, 7. and application. This model is influential. *E.g.*, we see the Connector Runtime as spanning the transport and session layers. 165, 233, 234, 237
- PDL Reowolf's Protocol Description Language is used to specify synchronous message communications in multi-party networks. The language was developed as part of the Reowolf project, and the formalisation and analysis of the language is the primary concern of Chapter 5. 7, 9, 13, 14, 143, 145–148, 150, 154, 157–166, 224, 231, 232, 235, 249, 252, 253
- PDL Runtime is a specification of runtime systems, that execute PDL protocols, contributed in Chapter 5. 143, 145, 147, 148, 153, 155, 158–162, 164, 165, 247
- Potassco The Potsdam Answer-Set Solving Collection (<https://potassco.org>) was developed at the University of Potsdam, and consists of a set of related answer-set solving tools, of which the Clingo solver is the most prominent. 230
- Prolog is an influential logic-programming language that is extensively applied to modelling and reasoning about systems. Prolog programs are comprised of declarative rules modelling steps of deductive inference, but some constructs (*e.g.*, the 'cut' operator) let programmers guide and optimise program evaluation. Many Prolog variants and tools exist, *e.g.*, the SWI Prolog interpreter at <https://www.swi-prolog.org>. 60, 62, 65, 72, 73, 94, 188
- PROMELA Process or Protocol Meta Language is a language used to model message-passing programs, and to verify their properties using the SPIN model-checker. 232, 237

Python is a high-level memory-managed scripting language. Traditionally, it was interpreted, and favoured a procedural style. But various tools have diversified its usages; *e.g.*, Python also offers object-oriented and functional language features, and Python programs can be statically type-checked and compiled. 41, 65, 165, 234

Reo is a synchronous, exogenous protocol specification language developed at CWI. Protocols are expressed as constraints on the data flowing between logical ports over logical instants. Reo is a major inspiration behind the Reowolf project relevant to Chapter 5. 9, 12, 13, 143–146, 162–164, 231, 236

Reowolf is a research project, whose website is <https://www.reowolf.net>, hosted at CWI and funded by the NGI zero PET, Pointer, and Assure funds. Reowolf developed a aimed to replace BSD sockets with connector in multi-party communications on the next-generation Internet, by lettings users specify network communications in Reo-like protocols. Reowolf is named in reference to Reo, the Beowulf. iii, 9, 12, 13, 143–146, 162–166, 226, 230–232, 235, 236, 246

REPL A read-evaluate-print loop is an interactive program that creates the abstraction of a persistent environment which the user can affect and inspect via commands or instructions. *E.g.*, the original eFLINT interpreter is a REPL, letting the user (interleavedly) trigger events, amend the specification, and inspect the current state and runtime history. 41

Rocq is the new name of the Coq language and prover. 13, 231

Rust is a statically-typed, systems programming language. Rust’s unique memory management system (‘the borrow checker’) performs most memory management at compile time. Rust is the implementation language of the BRANE services [VCB21], the Connector Runtime [EH24], our JustAct reproduction of BRANE in Chapter 6, and the SEASO and Slick interpreters in Chapter 4. 65, 135, 145, 160, 165, 170, 182, 185–187, 189, 190, 193–195, 221, 230, 236, 249

SAT The Boolean Satisfiability problem is fundamental in theoretical computer science and logic research: is a given logical formulae satisfiable by any model, *i.e.*, assignment of Boolean values to logical variables?. 95

SDN Software Defined Networks is a research area focusing on designing and exploiting specifications of computer networks in software, making the networks easier to (re)configure and analyse. 165

- SEASO is a specification language designed for the incremental and cooperative specification of data exchange systems. The language is named after an abbreviation of its statements: seal, emit (and so on). The design and application of SEASO is the focus of Chapter 3. 7, 9–12, 14, 26, 57, 60, 61, 65–75, 77, 79, 82, 84–89, 91–97, 99, 110, 125, 130–134, 139–141, 223, 236, 237, 244
- Slick is a SEASO variant developed in Chapter 4 for the case study in Chapter 6, by emphasising brevity, flexibility, and composability of program rules. 7, 9, 12, 99, 125, 133–135, 139–141, 169, 170, 180, 181, 186, 190–192, 194, 195, 198, 199, 205, 208, 211–214, 217, 223, 224, 226, 236
- SMT Satisfiability Modulo Theories extends SMT solving with specialised reasoning for particular theories that are common in practice, *e.g.*, integer arithmetic, finite text strings, arrays, and so on. 55, 237
- Soufflé is a Datalog-based logic programming language and reasoning engine that are often used for static system analysis, *e.g.*, of network topologies. 53
- SPIN The Simple PROMELA Interpreter simulates PROMELA programs and generates implementations in the C language, *e.g.*, which can then be model-checked. 235
- SSPDDP Secure Scalable Policy-enforced Distributed Data Processing preceded AMdEX, focused on logistical use cases, and led to the creation of eFLINT. 232
- SURF is an organisation that develops and maintains computer and network infrastructure supporting research in the Netherlands; accordingly, SURF is a member of AMdEX and EPI consortia. Their webpage is <https://www.surf.nl>. Their name abbreviates ‘cooperative university computer facilities’ (Dutch: *Samenwerkende Universitaire Rekenfaciliteiten*). 197, 202, 204, 207–213, 216, 218, 219
- Symboleo is a normative modelling and specification language focused on (business) contracts. 9, 64, 65, 94, 172
- TCP The Transmission Control Protocol is a popular two-party communication protocol in the transport layer of the OSI stack. The main idea is that a stream of bytes are incrementally transmitted from each party to their peer. TCP is often contrasted to UDP. 12, 237
- UDP The User Datagram Protocol is a popular two-party communication protocol in the transport layer of the OSI stack. The main idea is that datagrams (byte sequences) are exchanged in both directions without guarantees about their ordering or delivery. UDP is often contrasted to TCP. 12, 146, 170, 237

UML Unified Modelling Language is a ubiquitous modelling language for representing various facets of (software) systems graphically. Different kinds of UML diagram exist for representing different facets; *e.g.*, component diagrams visualise systems as decomposed into loosely-coupled (communicating) components, while sequence diagrams lay the messages communicated between discrete agents over time. 95, 111

UvA The University of Amsterdam (Dutch: *Universiteit van Amsterdam*) is a Dutch research university. The CCI group in the UvA informatics institute hosted me during my time as a PhD student (Dutch: *promovendus*). iii, iv, 3, 11, 58, 229, 230, 233, 238

XACML Extensible Access Control Markup Language [ANP<sup>+</sup>03] is a software framework and EDSL in XML for specifying access control, *i.e.*, conditional permission of access to resources. The work is collated in standard documents; version 3 is available at <https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html>. The XACML framework is influential, *e.g.*, many systems have been designed around XACML roles such as ‘policy decision point’ and ‘policy administration point’. 4, 64, 171

XML Extensible Markup Language is a system for structured textual information often used to embed EDSLs and for transport over networks. 235, 238

### A An Account of the Situation, Event, and Fluent Calculi

Here, we briefly overview three related logical theories for dynamic systems which have influenced eFLINT, so their influence is evident in our state traces in Section 2.4.

Many logical theories have been developed to model and evaluate dynamic systems. These theories are inter-related, and have influenced the foundations and usage of normative specifications. We focus on the *situation*, *event*, and *fluent* calculi and their (partially overlapping) concepts. These three theories formalise essentially comparable ideas. For example, *states* are related to *fluents* by the *holds* relation. Intuitively, fluents are Boolean variables whose truth can be added and removed to/from states. These effects arise from *events* (sometimes called *actions*). Each theory represents a unique trade-off between which concepts are fundamental, which concepts can be (elegantly) expressed, and which queries can be (efficiently) evaluated. In its own way, each theory addresses the *frame problem* [Ams88]: how can the values of fluents be efficiently expressed and computed if, in the worst case, each event affects each fluent? Naïve solutions generally exhibit combinatorial explosion. Each of these theories builds a unique solution by starting from the same practical assumption [Pea94]: each event typically affects only a small subset of the large universe of fluents.

**Situation Calculus** John McCarthy proposed the situation calculus for expressing and reasoning about (laws regulating) causal events in 1963 [McC63], but more recent writings [McC02] are more accessible. Reasoning is oriented around *situations*: lists of prior events since the unique and trivial *initial* situation. Thus, each situation has a unique, *resulting* state. Terms and formulae are often defined via induction on the structure of situations. For example, fluents are evaluated by unfolding the situation and reducing (the terms defining) the effects of prior events. Situations afford the robust expression and comparison of alternative actions because each situation fixes its own history of prior events. However, the discrete structure of situations made it cumbersome for modelling continuous time, and to change events across situations.

Reasoning is also complicated by the presence of *transpositions*: the same states resulting from different situations.

**Event Calculus** The event calculus addressed the pragmatic challenges that arose in certain applications of the situation calculus. Events and states are inter-related indirectly: each occurs at a *time* and time(point)s are ordered. Each fluent is true at time  $t$  if, until  $t$ , its truth was added more recently than it was removed. This approach makes it natural to orthogonalise the definitions of different events and their effects on fluents, and it neatly generalises to continuous time or simultaneous events. Naturally, the event calculus is more cumbersome in cases where situations are the most natural representation of knowledge. For example, to support the common problem of *planning* the actions of AI agents to reach states with given properties, many steps of reasoning about the fluents and events are often needed.

**Fluent Calculus** The fluent calculus is explicitly motivated by a strong interpretation of the frame problem: ‘to not apply separate inference steps for each unaffected piece of knowledge’ [Thi00], *i.e.*, both the *expression* and the *computation* of events and states should be unencumbered by unrelated events and unaffected fluents. The main idea is to identify states via the fluents themselves, such that effects are defined as state-transformations, and fluents are evaluated by simplifying these transformed state-terms. Intuitively, the evaluation of affected fluents is guided by the syntax of terms defining the effects themselves. Like the situation calculus, the fluent calculus focuses on orthogonalising unrelated events: a *successor state* axiom defines the post-state of an action as the pre-state with the desired fluents added and removed [ST06]. An example drawback is that diagnosing the events resulting in a given state generally requires many inference steps and the comparison and transformation of many state-terms, which may be large and complex.

## B Example 2.1 Translated from eFLINT to Clingo in Full

The following Clingo ruleset is the result of translating the running example input eFLINT specification and scenario Example 2.1. What is shown here is unaltered, *i.e.*, reproducible via artefact [MEvB25], except that we have massaged the spacing of terms, and preceded (clusters of) Clingo rules with explanatory comments.

Note that each rule in Definitions 2.1 and 2.4 with  $N$  conjunct consequents is expressed as  $N$  Clingo rules. Also note that the scenario rules have an unnecessary (but harmless) `state(s)` antecedent as an artefact of our translation of scenarios re-using the translation function for specifications, where this antecedent is appropriate.

----- Part 1/4: fixed state trace semantic rules -----

```
% (inter)state rules and constraints
state(1).
state(S) :- in(F,S).
state(S - 1) :- state(S) ; 1 < S.
:- state(S) ; not 0 < S.

% event calculus style inertia
in( F , S + 1) :- in((add,F),S) ; not in((rem,F),S) ; state(S + 1).
in((add,F),S + 1) :- in((add,F),S) ; not in((rem,F),S) ; state(S + 1).
```

----- Part 2/4: fixed core eFLINT semantic rules -----

```
% UPPER top left
in((add, created(I)),S) :- in((create,I),S).
in((rem, terminated(I)),S) :- in((create,I),S).

% UPPER top right
in((rem, created(I)),S) :- in((terminate,I),S) ; not in((create,I),S).
in((add, terminated(I)),S) :- in((terminate,I),S) ; not in((create,I),S).

% UPPER left
in((rem, created(X)),S) :- in((obfuscate,X),S) ;
    not in((terminate,I),S) ; not in((create,X),S).
in((rem, terminated(I)),S) :- in((obfuscate,X),S) ;
    not in((terminate,I),S) ; not in((create,X),S).

% MIDDLE left
in((holds,I),S) :- in((created,I),S).

% MIDDLE bottom right
in((holds,I),S) :- in((derived,I),S) ;
    not in((suppressed,I),S) ; not in((terminated,I),S).

% MIDDLE top right
in((enabled,I),S) :- in((holds,I),S) ; not in((suppressed,I),S).

% LOWER right
in((dutyViolation,I),S) :- in((violated,I),S) ; in((enabled,I),S).

% LOWER left
in((actViolation,I),S) :- in((actTrigger,I),S) ; not in((enabled,I),S).
```

----- Part 3/4: translated specification rules -----

```
% Act access ... Derived from (Foreach user,controls,instant:
% access(user,controls.dataset,instant))
in((derived,access(user(D),dataset(B),instant(C))),S) :- state(S) ;
    in((enum,user(D)),S) ; in((enum,controls(user(A),dataset(B))),S) ;
    in((enum,instant(C)),S).
```

```

% Act access ... Creates (Foreach controls:
%   must_notify(user,controls.controller,access(user,dataset,instant),instant + 10)
%   Where controls.dataset == dataset)
in((create,must_notify(user(E),user(A),access(user(E),
                    dataset(C),instant(D)),instant((D + 10)))),S) :-
    state(S) ; dataset(B) = dataset(C) ; in((enum,controls(user(A),dataset(B))),S) ;
    in((trigger,access(user(E),dataset(C),instant(D))),S).

% access (implicitly) defined as an infinite type
in((enum,access(user(C),dataset(A),instant(B))),S) :- state(S) ;
    in((holds,access(user(C),dataset(A),instant(B))),S).

% notify defined as an action type
in(actTrigger(access(user(C),dataset(A),instant(B))),S) :- state(S) ;
    in((trigger,access(user(C),dataset(A),instant(B))),S).

% 'inbuilt' actor (implicitly) defined as an infinite type
in((enum,actor(A)),S) :- state(S) ; in((holds,actor(A)),S).

% Fact controls ... Derived from (Foreach dataset:
%   controls(user("Admin"),dataset)
%   Where Not(Exists user: user != user("Admin") && controls(user,dataset)))
in((derived,controls(user("Admin"),dataset(A))),S) :- state(S) ; not 0 < #count{
    user(B) : #true ,not user(B) = user("Admin")
    ,in((holds,controls(user(B),dataset(A))),S) ,in((enum,user(B)),S) } ;
    in((enum,dataset(A)),S).

% controls (implicitly) defined as an infinite type
in((enum,controls(user(A),dataset(B))),S) :- state(S) ;
    in((holds,controls(user(A),dataset(B))),S).

% Fact dataset ... Derived from (Foreach controls: controls.dataset)
in((derived,dataset(B)),S) :- state(S) ; in((enum,controls(user(A),dataset(B))),S).

% dataset (implicitly) defined as an infinite type
in((enum,dataset(A)),S) :- state(S) ; in((holds,dataset(A)),S).

% elapsed (implicitly) defined as an infinite type
in((enum,elapsed(instant(A))),S) :- state(S) ; in((holds,elapsed(instant(A))),S).

% instant (implicitly) defined as an infinite type
in((enum,instant(A)),S) :- state(S) ; in((holds,instant(A)),S).

% 'inbuilt' int (implicitly) defined as an infinite type
in((enum,int(A)),S) :- state(S) ; in((holds,int(A)),S).

```

```

% Duty must_notify ...   Violated when Holds(elapsed(deadline))
in((violated,must_notify(user(G),user(E),access(D,C,B),instant(A))),S) :-
    state(S) ; in((holds,elapsed(instant(A))),S) ;
    in((enum,must_notify(user(G),user(E),access(D,C,B),instant(A))),S).

% instant (implicitly) defined as an infinite type
in((enum,must_notify(user(F),user(D),access(C,B,A),instant(E))),S) :- state(S) ;
    in((holds,must_notify(user(F),user(D),access(C,B,A),instant(E))),S).

% Act notify ...   Derived from (Foreach must_notify:
%   notify(must_notify.user,must_notify.controller,must_notify))
in((derived,notify(user(A),user(B),must_notify(user(A),
    user(B),access(E,D,C),instant(F))))),S) :-
    state(S) ; in((enum,must_notify(user(A),user(B),access(E,D,C),instant(F))),S).

% Act notify ... Terminates must_notify
in((terminate,must_notify(D,C,B,A)),S) :- state(S) ;
    in((trigger,notify(user(J),user(E),must_notify(D,C,B,A))),S).

% notify (implicitly) defined as an infinite type
in((enum,notify(user(F),user(A),must_notify(E,D,C,B))),S) :- state(S) ;
    in((holds,notify(user(F),user(A),must_notify(E,D,C,B))),S).

% notify defined as an action type
in(actTrigger(notify(user(F),user(A),must_notify(E,D,C,B))),S) :- state(S) ;
    in((trigger,notify(user(F),user(A),must_notify(E,D,C,B))),S).

% 'inbuilt' string (implicitly) defined as an infinite type
in((enum,string(A)),S) :- state(S) ; in((holds,string(A)),S).

% Fact user ... Derived from (Foreach controls: controls.controller)
in((derived,user(A)),S) :- state(S) ; in((enum,controls(user(A),dataset(B))),S).

% user defined as an action type
in((enum,user(A)),S) :- state(S) ; in((holds,user(A)),S).

```

----- Part 4/4: translated scenario rules -----

```

% the first four effects, labelling the first four transitions, respectively
in((create,          dataset("X-Rays")          ),1) :- state(S).
in((create,controls(user("Amy"),dataset("X-Rays") ),2) :- state(S).
in((create,          user("Bob")                ),3) :- state(S).
in((create,          instant(9)                  ),4) :- state(S).
in((create, access(user("Bob"),dataset("X-Rays"),instant(9))),5) :- state(S).

% the terminus of (final transition in) the state trace is the 5th state
state(5).

```

## C One Seaso Denotation Computed in Full

We clarify the semantics of SEASO by detailing the computation of the denotation of well-formed program  $s^*$ . For brevity in this section, we 1. use  $a_{i-j}$  to abbreviate atom set  $\{a_i, a_{i+1}, a_{i+2}, \dots, a_j\}$ , 2. let  $\mathcal{A} = a_{1-3}$ , because all supersets of this  $\mathcal{A}$  yield the same result, and 3. omit the fixed program  $s^*$  from all terms, for example, we write  $(\langle s^*, \emptyset \rangle \Longrightarrow a_{1-3})$  instead as  $(\emptyset \Longrightarrow a_{1-3})$ . Let  $s^*$  consist of the following rules:

id	rule	in natural language
$r_1$	$rule(a_1, a_{3-3}, a_{2-2})$	$a_1$ is true if $a_3$ is true and $a_2$ is false.
$r_2$	$rule(a_2, \emptyset, a_{1-1})$	$a_2$ is true if $a_1$ is false.
$r_3$	$rule(a_3, \emptyset, \emptyset)$	$a_3$ is true.

The following shows the use of meta-rule BIG to prove membership of elements in  $(\Longrightarrow)$  sufficient to compute the denotation. The premise of each meta-rule is another proof tree, encoding a terminating path of  $(\longrightarrow)$  steps. We omit these proofs, but we suggest their contents by labelling each  $(\longrightarrow)$  with the relevant rule in  $r_{1-3}$ , *i.e.*, the rule bound to meta-variable  $s_r$  when applying SMALL.

$$\frac{\langle a_{1-3}, \emptyset \rangle \xrightarrow{r_3} \langle a_{1-3}, a_{3-3} \rangle \xrightarrow{r_2} \langle a_{1-3}, a_{2-3} \rangle \xrightarrow{r_1} \langle a_{1-3}, a_{1-3} \rangle}{a_{1-3} \Longrightarrow a_{1-3}} \text{ BIG}$$

$$\frac{\langle \emptyset, \emptyset \rangle \xrightarrow{r_3} \langle \emptyset, a_{3-3} \rangle}{\emptyset \Longrightarrow a_{3-3}} \text{ BIG}$$

$$\frac{\langle a_{1-2}, \emptyset \rangle \xrightarrow{r_3} \langle a_{1-2}, a_{3-3} \rangle \xrightarrow{r_2} \langle a_{1-2}, a_{2-3} \rangle \xrightarrow{r_1} \langle a_{1-2}, a_{1-3} \rangle}{a_{1-2} \Longrightarrow a_{1-3}} \text{ BIG}$$

The following shows a proof tree whose results determine elements matching  $(i \rightsquigarrow \dots)$  for  $i$  in 0–4. These suffice to compute the denotation  $\langle a_{3-3}, a_{1-2} \rangle$ , read as ‘ $a_3$  is true, and  $a_{1-2}$  are unknown’. We use  $(\neg)$  to denote each set complement, *i.e.*,  $(A_1 \neg A_2) \leftrightarrow (A_1 = \mathcal{A} \setminus A_2)$ . Finally, for brevity, we show the three premises of each BIG application  $\{n \rightsquigarrow A_1, A_1 \neg A_2, A_2 \Longrightarrow A_3\}$  as the chain  $(n \rightsquigarrow A_1 \neg A_2 \Longrightarrow A_3)$ .

$$\begin{array}{l} \text{ALT BASE} \\ \text{ALT STEP} \end{array} \frac{\overline{0 \rightsquigarrow \emptyset \neg a_{1-3} \Longrightarrow a_{1-3}}}{\text{ALT STEP} \frac{\overline{1 \rightsquigarrow a_{1-3} \neg \emptyset \Longrightarrow a_{1-3}}}{\text{ALT STEP} \frac{\overline{2 \rightsquigarrow a_{1-3} \neg a_{2-3} \Longrightarrow a_{1-3}}}{\text{ALT STEP} \frac{\overline{3 \rightsquigarrow a_{1-3} \neg \emptyset \Longrightarrow a_{3-3}}}{4 \rightsquigarrow a_{3-3}}}}}$$

$$\frac{\begin{array}{l} \neg ill(s^*) \\ 4 \% 2 = 0 \\ 4 \rightsquigarrow a_{3-3} \end{array}}{\text{DENOTATION}} \frac{\overline{2 \rightsquigarrow a_{3-3}}}{\overline{3 \rightsquigarrow a_{1-3}}}$$

$$s^* \models (a_{3-3}, (a_{1-3} \setminus a_{3-3}))$$

The intuition behind this linear proof tree is clarified by representing the ALT STEP instances as steps in a path through atom sets, one  $(A_1 \rightarrow A_2 \Longrightarrow A_3)$  at a time. The box encloses the beginnings of an ALT STEP cycle of period 2, which repeats forevermore; this is the alternating fixpoint, which alternates between atom sets  $a_{3-3}$  and  $a_{1-3}$ . Note how these determine the denotation.

$$\emptyset \rightarrow a_{1-3} \Longrightarrow \boxed{a_{1-3} \rightarrow \emptyset \Longrightarrow a_{3-3} \rightarrow a_{1-2} \Longrightarrow a_{1-3}} \rightarrow \dots$$

## D Formalising Extra Data Exchange Archetypes in Seaso

Here, we give more examples formalising data exchange archetypes shown in Figure 3.1. For brevity, listings in this section omit prefix `defn orgA(agent, context).` `orgB(agent, context).` `orgC(agent, context).` and the suffix `seal computer. functionProvider. consumer. parameterProvider.`

**Fragment D.1** (the ‘sharing results’ archetype).

```
rule computer(A,C), parameterProvider(A,C), functionProvider(A,C) :- orgA(A,C).
    consumer(A,C) :- orgB(A,C).
```

**Fragment D.2** (the ‘infrastructure as a service’ archetype).

```
rule consumer(A,C), parameterProvider(A,C), functionProvider(A,C) :- orgA(A,C).
    computer(A,C) :- orgB(A,C).
```

**Fragment D.3** (the ‘sharing data via a trusted third party’ archetype).

```
rule parameterProvider(A,C), :- orgA(A,C).
    consumer(A,C), functionProvider(A,C) :- orgB(A,C).
    computer(A,C) :- orgC(A,C).
```

**Fragment D.4** (the ‘sharing code via a trusted third party’ archetype).

```
rule parameterProvider(A,C), functionProvider(A,C) :- orgA(A,C).
    consumer(A,C) :- orgB(A,C).
    computer(A,C) :- orgC(A,C).
```

Our approach includes the context parameter such that agent-roles may be contextualised. A motivating example is a system using a combination of archetypes. For example, the following formalises a generalisation of archetypes *Sharing data via TTP* and *Share code and data via TTP*. Intuitively, roles are contextualised by the agent playing the `functionProvider` role.

**Fragment D.5** (a combination of two archetypes). This fragment creates a new archetype by combining the existing archetypes ‘sharing data via a trusted third party’ and ‘Share code and data via a trusted third party’. Intuitively, roles are contextualised by the agent playing the `functionProvider` role.

```
defn context(agent).
rule parameterProvider(A,C)      :- orgA(A,C).
   consumer(A,C)                 :- orgB(A,C).
   computer(A,C)                 :- orgC(A,C).
   functionProvider(A,context(A)) :- context(A).
```

## E Reowolf Terms in Chapter 5 vs. the Formalism in Coq

The definitions in this chapter correspond to those in a supplementary artefact: a machine-checked Coq specification. The repository at <https://zenodo.org/records/14936561> includes the Coq file itself `reowolf_formalism.v`, as well as instructions for installing Coq locally or via a Docker image.

Figure 1 shows the *parameters* of this formalisation. As our definitions are otherwise constructive, these separate our specification from an (executable) implementation. The parametrisation of  $\mathcal{P}$  and  $\mathcal{V}$  is desirable, affording the choice of any concrete port- and variable-types, *e.g.*, in building a particular implementation. The remaining three parameters can be understood as *assumptions*, scoping our formalism. Equivalently, these properties are assumed but have no proofs yet. The first two assumptions constrain the choice of  $\mathcal{P}$  and  $\mathcal{V}$ ; they must form *setoids*, because our proofs rely on being able to decide (in)equality of port pairs and variable pairs. The final assumption requires the refinement of the memory-storage to some finite map structure, for example, implemented via Coq’s inbuilt association lists or AVL trees.

Figure 2 shows the correspondences between the definitions in Chapter 5 to those in the artefact, *e.g.*, to help readers inspect the relevant Coq definitions.

in natural language	in Section 5.3 notation	definition in Coq
the port type	$\mathcal{P}$	port in line 77
the variable type	$\mathcal{V}$	var in line 77
decidable port equality	$\forall p p' : \mathcal{P}, p = p' \vee p \neq p'$	eq_dec_port in line 78
decidable var. equality	$\forall v v' : \mathcal{V}, v = v' \vee v \neq v'$	eq_dec_var in line 79
decidable mem. equality	$\forall \sigma \sigma' : \Sigma, \sigma = \sigma' \vee \sigma \neq \sigma'$	eq_dec_memory in line 146

Figure 1: Parameters of our Reowolf formalism [ELH25], where each is marked by the `Parameter` keyword.

Term in Chapter 5	Term in the Coq Artefact
• $\mathcal{D}$ in Definition 5.1	data in line 71
• $\mathcal{P}$ in Definition 5.1	port in line 77
• $\mathcal{V}$ in Definition 5.3	var in line 77
• $\mathcal{M}$ in Definition 5.1	msg in line 82
• $\leq$ in Definition 5.7	leq_msg in line 93
• $\mathcal{E}$ in Figure 5.2	expression in line 114
• $\mathcal{S}$ in Figure 5.2	statement in line 121
• $\Sigma$ in Definition 5.3	memory in line 142
• $\mathcal{W}$ in Definition 5.3	worker in line 152
• <i>eval</i> declared in the caption of Figure 5.3	eval in line 177
• $\Delta$ in Definition 5.4	synchronicity in line 189
• <i>update</i> in Figure 5.3	update in line 223
• Definition 5.11	growing_msglist in line 572
• Definition 5.13	constant_msglist in line 585
• $W \xrightarrow{m} W'$ in Definition 5.5	sync in line 708
• $W \xrightarrow{m} W'$ in Definition 5.5	async in line 717
• Lemma 5.1	asynchronous_step_determinism in line 950
• $W \not\rightarrow W'$ in Section 5.3.3	apath in line 975
• Definition 5.9	round in line 988
• Definition 5.10	round_final_m in line 993
• Lemma 5.2	per_growing_round_a_cst in line 1677
• Lemma 5.3	cst_confluence in line 1468
• Theorem 5.1	round_determinism in line 1712
• Theorem 5.2	compositionality in line 1950
• Theorem 5.3	decompositionality in line 2090
• Definition 5.14	offered in line 2134
• Definition 5.15	constructive_round in line 2170
• Lemma 5.4	const_constructability in line 2454
• $2^{\mathcal{S}}$ in Figure 5.2	protocol in line 2512
• Definition 5.12	accepts in line 2537
• Definition 5.16	constructs in line 2541
• Theorem 5.4	acceptance_vs_construction in line 2735
• PDL Runtime $C$ and signature Definition 5.17 in Section 5.4.1	PdLRuntime record with fields {config, start, proto, inject, run} in line 2752
• Property 5.1	initially_trivial_protocol in line 2763
• Property 5.2	injection_composes in line 2767
• Property 5.3	soundness in line 2774
• Property 5.4	completeness in line 2788
• <i>Silent PDL Runtime</i> in Section 5.4.2	silent_pdl_runtime in line 2763
• Proofs of Properties 5.1 to 5.3 and inverse of Property 5.4 for the silent PDL Runtime in Section 5.4.2 (only their existence is claimed)	silent_initially_trivial_protocol at line 2817, silent_injection_composes at line 2824, silent_soundness at line 2840, and silent_incompleteness at line 2850 (resp.).

Figure 2: Corresponding terms between Chapter 5 and its artefact [ELH25].

## F Gentle Introduction to (Our Reowolf Formalism in) Coq

Here, we improve the accessibility of our Coq formalisation by briefly characterising Coq formalisations in general, and ours in particular. This section is written to be accessible to readers that are already somewhat familiar with functional programming, but perhaps unfamiliar with the Coq language, dependently-typed functional programming, or interactive proof assistants. Our goal is to demystify our formalism for these readers. More complete explanations and demonstrations of Coq are plentiful online; we can personally vouch for the quality of the first two volumes of the *Software Foundations* book series [PCG<sup>+</sup>23a, PCG<sup>+</sup>23b] by Pierce et al.

### Definitions, Terms, Types, and Functions

Each Coq program or specification is a sequence of statements. The most fundamental statements define new terms. In Example F.1, `:=` separates the identifier `data` from the value `nat`. So this statement asserts the definitional equality between `data` and Coq’s standard natural number type `nat` (see Section 5.2). As is often the case, the type of the defined type is omitted, because Coq can infer it in context. Otherwise, or for added readability, they can be stated explicitly. We would insert `: set` after `data` in Example F.1. Generally, terms can be annotated with type assertions in this manner, *e.g.*, `: set` can also be inserted after `nat` in Example F.1.

`set` is the type of ‘computable data types’, which is distinguished from `Prop`, the type of ‘propositions’, which we address soon. For example, functional programs are often written mostly in `set`, and reasoned about in `Prop`, and Haskell and OCaml code is generated by erasing all usages of `Prop`. Also, `Type` generalises `set` and `Prop`.

**Example F.1** (let `data` be the natural numbers).

```
Definition data := nat.
```

Terms are introduced with given types but *without values* using the `Parameter` keyword. Example F.2 introduces `port` and `var`, the ports  $\mathcal{P}$  and variables  $\mathcal{V}$  in Section 5.2. Subsequent definitions can use these terms, but they are opaque except for their type. For example, they can be passed into functions expecting `set`. The idea is that these are arbitrary in our specification; our definitions and proofs never ‘look inside’ variables and ports, so users can choose any members of `set` at all.

**Example F.2** (take arbitrary data types, `port` and `var`).

```
Parameters port var : Set.
```

Example F.3 defines `msg : set`, which corresponds to the ‘message maps’ or ‘messages’ type  $\mathcal{M}$  that is introduced in Section 5.2. Evidently, each member of `msg` is a function, which evaluates to an *optional data*, where `option` is defined in the Coq prelude, and produces a new type when applied to a type, *i.e.*, it is a polymorphic

type that is comparable to `Maybe` in Haskell or `Option` in Rust. Example F.3 also defines `write_msg`, which is a pure function, but which models the mutation of `msg` elements. The definition specifies how `write_msg m p d` terms can be normalised, *i.e.*, simplified to a unique term. The new term relies on `eq_dec_port` deciding whether or not the two given ports, named `p` and `p'`, are (syntactically) equivalent.

**Example F.3** (message maps and writing messages).

```

Definition msg := port -> option data.
Definition write_msg (m: msg) (p: port) (d: data) : msg :=
  fun p' => if eq_dec_port p p' then Some d else m p'.

```

## Inductive Types and Constructor Functions

Example F.4 provides an *inductive* definition of two types: `bin_op` and `expression`. Definitions such as these introduce a ‘type’ term  $\tau$  alongside a closed set of *constructors*, either of type  $\tau$ , or functions to (functions to ...) type  $\tau$ . Like other functional programming languages like Haskell, (only) constructor do not *normalise* when applied to arguments. For example, although `Const : data -> expression` is a function, `Const 2` is in normal form: it cannot be simplified further. Note that constructor `Not` (and others) construct expressions from (sub) expressions; `expression`, like many inductive types, has infinitely many members, but each is some finite term. This is an important distinction from, say, Haskell; Coq guarantees that every term of every inductively defined type takes only finitely many step to normalise. Terms without this property are introduced with other keywords like `Coinductive`, but our formalisation does not use these, so we consider them no further.

Note that terms denoting (finite) terms of inductive types can be arbitrarily large, but still, they are necessarily normalising. For example, using `write_msg` from Example F.3, it is clear how to build arbitrarily large but finite terms of type `msg`, *e.g.*, matching `write_msg (write_msg (write_msg ( ... ) p3 d3) p2 d2) p1 d1`, by unfolding and simplifying each of its function application subterms, in turn.

Example F.4 in particular formalises the abstract syntax of PDL expressions, denoted as  $\mathcal{E}$  in Figure 5.2, via an embedding as the Coq type `expression`. Consequently, each element of type (uniquely) denotes a particular PDL expression. For example, `BinOp Add (Read v1) (Read v2)` denotes the sum of data read from variables `v1` and `v2`.

**Example F.4** (PDL expression syntax).

```

Inductive bin_op      := Add | Sub | Mul | Div | Mod | Eq | And | Or.
Inductive expression := Const (d : data)
  | Read (v : var)
  | BinOp (o : bin_op) (ex1 ex2 : expression)
  | Not (ex : expression).

```

Almost nothing is ‘intrinsic’ in Coq; even the ubiquitous natural numbers (`nat`), polymorphic lists (`list`) and options (`option`) are user-definable. Example F.5 shows their full definitions in the standard library, which are retrieved by letting the Coq proof assistant evaluate the queries `Print nat`, `Print option`, and `Print list`.

**Example F.5** (standard definitions of some familiar essentials).

```
Inductive nat : Set := 0 : nat | S : nat -> nat.
Inductive option (A : Type) : Type := Some : A -> option A | None : option A.
Inductive list (A : Type) : Type := nil : list A | cons : A -> list A -> list A.
```

## Terms of Types vs. Proofs of Propositions

Example F.6 demonstrates the definition of a *property* (of `msg` pairs called `m1` and `m2`), *i.e.*, a relation over `msg`. Evidently, this property is defined as a function over `m1` and `m2` to `Prop`, which is the type of (*logical*) *propositions*. Example F.6 demonstrates how these two worlds coincide in Coq, which emphasises the Curry-Howard correspondence: types correspond to propositions, and their members correspond to proofs! The constructs of Coq are intentionally designed to hammer this correspondence home. For example, each `x = y` is syntactic sugar for `eq x y`, where `eq` is an inductively-defined type that is essentially similar to `expression` in Example F.4. So `leq_msg` ultimately returns terms in `eq`, the proposition that two terms are (syntactically) equivalent, whose values are proofs of the proposition. Consequently, Example F.6 defines each `leq_msg m1 m2` as the proposition that, for each port `p` and data `d`, if the data `m1 p` is `d`, then the data `m2 p` is also `d`. In natural language, we might say ‘proving `leq_msg m1 m2` is proving that `m1` has a subset of the messages in `m2`’.

To understand Example F.6, the final detail to explain is `forall`. This primitive leverages Coq’s *dependent* type foundations. Conceptually, this language feature is orthogonal to the Curry-Howard correspondence, but in practice, it enables the expression of powerful properties. Precisely, `forall (a : A), B` is identical to `A -> B` in that it defines a function from `A` to `B`, but only the former *names* the arbitrary input value, such that it can be subsequently referred to. In the case of Example F.6, `forall p d, ...` abbreviates `forall (p: port), forall (d: data), ...`, and binds `p` and `d` for use in the term `m1 p = Some d -> m2 p = Some d`, whose root is `->`: simultaneously a function transformaing any proof of `m1 p = Some d` to a proof of `m2 p = Some d` and a *logical implication* from left to right; if the former holds, then the latter holds.

When using Coq, it is useful to dance between these dual views on terms and types. For example, `leq_msg : msg -> msg -> Prop` affords the logical interpretation of `leq_msg` as ‘a property of `msg` pairs’. But `leq_msg` is also a Coq function, much like any other, so any term of type `leq_msg m1 m2 p : data -> Prop` is a function that is applicable to any data, which will produce a proof of a proposition as output. With this view,

indeed, *every* definition proves something, but many are uninteresting. For example, it is accurate to interpret Example F.1 as proving ‘an element of `set` exists’.

**Example F.6** (the less-than relation over message maps).

```
Definition leq_msg (m1 m2 : msg) : Prop :=
  forall p d, m1 p = Some d -> m2 p = Some d.
```

Example F.7 showcases a simple usage of Coq’s `Notation` statements. Formally, these do very little; they simply introduce new syntactic sugar for Coq expressions. However, these kinds of notations are invaluable in practice, because they let Coq terms even more closely mimic mathematical and logical conventions. For example, note how Example F.6 relies on Coq’s standard `=` notation for the `eq` proposition. Example F.7 lets each `m1 <= m2` term more intuitively notate the term `msg_eq m1 m2`.

**Example F.7** (defining a familiar notation for less-than).

```
Notation "m1 <= m2" := (leq_msg m1 m2) (at level 70).
```

Example F.8 defines `leq_msg_refl`, whose type encodes the (interesting) property that the relation `<=` is *reflexive*. The type of `leq_msg_refl` encodes reflexivity in the usual way: it relates every member of `msg` to itself. Equivalently, `leq_msg_refl` defines a function which, given any `m : msg`, returns a proof of the proposition `m <= m`.

This definition is formulated as a *lemma* using the `Lemma` keyword. Coq offer `Theorem` and `Corollary` as stylistic alternatives to `Lemma`. All of these are essentially similar to `Definition`: they define a term of a given type. The only difference from `Definition` is that `Lemma` accepts a type and then enters *proof mode*, where Coq accepts a radically different input language from what we have seen so far, where the programmer details the application of *tactics*. For example, `unfold` is a tactic, applied to the term `msg_leq` via the `<=` notation. Ultimately, tactics let programmers guide Coq into the definition of a term (*i.e.*, a proof) via abstract prompting that are designed to mimic the argumentation of mathematical proofs. Coq manages a (*proof*) *context*, containing a set of *premises* (typed terms, *i.e.*, proven propositions) and remaining proof goals (obligations to prove propositions, *i.e.*, typed ‘holes’). Most tactics are named to suggest the mathematical argumentation they mimic. For example, the `assumption` tactic simply completes the current goal  $\tau$  if there is any  $\tau$ -type premise already in context (‘the goal follows trivially from a premise’) and `destruct` breaks one goal into many, one for each possible of constructor of a given term.

The tactics proving Example F.8 can be roughly understood as 1. rewriting the premise `m <= m` to `forall p d, m p = some d -> m p = some d` by unfolding the notation `<=` and definition `msg_leq`; 2. removing `forall m p d H` from the goal and adding premises `m : msg`; `p : port`; `d : data`; `H : m p = some d`, and finally; 3. reusing `H` to prove the only goal.

**Example F.8** (a proof of reflexivity demonstrating basic tactics).

```
Lemma leq_msg_refl: forall m, m <= m.
Proof. unfold "<=" . intros. assumption. Qed.
```

Where the proof term `leq_msg_ref1` is build by the Coq proof assistant, via the programmer's guidance with tactics, `leq_msg_ref1'` in Example F.9 defines an identical term 'by hand' via a definition. In fact, `Definition leq_msg_ref1' := leq_msg_ref1` would have achieved the same result! Here, the handcrafted version is smaller and simpler. But in general, in skilled hands, small and intuitive sequences of tactics can result in monstrously complex proof terms. This is sometimes unavoidable; it is easy to express succinct properties whose proofs require subtle argumentation in detail.

**Example F.9** ('hand-defining' a proof term).

```
Definition leq_msg_ref1' : forall m, m <= m :=
  fun m p d (H: m p = Some d) => H.
```

Our formalism includes a great many lemmas proving all sorts of propositions. Their proofs are often difficult to understand (even for us). But we rely on a quality of Coq that is often implicitly understood between its users, but which should be made explicit to newcomers: *it often suffices to understand the proven proposition*. In other words, we are primarily concerned with formulating the propositions themselves clearly, because they must often be understood by human readers. But seldom do readers need to understand the argumentation behind a proof.

For example, `asynchronous_step_determinism` in Example F.10 formalises and proves Lemma 5.1, whose importance to PDL is explained in Section 5.3.2. Primarily, we focus on clarifying the property: whenever workers `ws` can asynchronous step to `ws'` with some messages (`async ws m ws'`), then the step is preserved by adding to the messages (`async ws m ws'` where `m' <= m`). Secondly, the proof itself (between `Proof.` and `Qed.`) walks the reader through the argumentation. Here, the proof is formatted to reveal its structure, but we rely on Coq to organise the proof context, expecting readers to do likewise. For example, without Coq, it is not obvious that `H1` identifies the premise `async ws m ws'`. But Coq makes this premise clear by tracking the context.

**Example F.10** (a more realistic proof of a useful PDL property).

```
Lemma asynchronous_step_determinism:
  forall ws m ws' m',
    m <= m'      ->
    ws <> ws'    ->
    async ws m ws' ->
    async ws m' ws'.

Proof.
  intros. induction H1.
  - pose proof (update_inv m m' w w'). apply Async0. apply H2.
    + apply (list_neq _ _ _ eq_dec_worker) in H0.
      destruct H0; cbn in H0; [ | contradiction ] . apply Some_fequal in H0. apply H0.
    + apply H.
    + apply H1.
  - apply AsyncN. apply IHasync.
```

```
+ apply H.  
+ apply (list_neq _ _ _ eq_dec_worker) in H0.  
  cbn in H0. destruct H0; [ contradiction | apply H0 ].  
Qed.
```

Ultimately, our formalism is designed to cater to two kinds of readers. The first kind is expected to be in the vast majority: readers who strive to understand the properties being proven, in isolation, and, trusting Coq to check that the proofs are correct, without understanding the proofs themselves. The second kind of reader works to understand how the proofs are composed, and how the various properties relate to one another. For example, we expect to act as this kind of reader in the future, if we extend our formalism to prove new properties, or to enrich PDL with new features. Fortunately, proofs can be used to build other proofs without understanding them. For example, `asynchronous_step_determinism` (in Example F.10) is used in the proof of `lift_cst_aphath_msg`, which proves a crucial lemma about *asynchronous paths* (denoted  $\rightarrow^*$  in Section 5.3.3) which underpins our proof of Theorem 5.1.

---

## Bibliography

- [AAA<sup>+</sup>20] Frank M Aarestrup, Abdullah Albeyatti, W John Armitage, Charles Auffray, Luca Augello, Rudi Balling, Nora Benhabiles, Guido Bertolini, Jan G Bjaalie, Michaela Black, et al. Towards a european health research and innovation cloud (hric). *Genome medicine*, 12:1–14, 2020.
- [ABB<sup>+</sup>18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15. ACM, 2018.
- [AGNvdT13] Giulia Andrighetto, Guido Governatori, Pablo Noriega, and Leendert W. N. van der Torre, editors. *Normative Multi-Agent Systems*, volume 4 of *Dagstuhl Follow-Ups*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.
- [AJT19] Andreas Arvidsson, Moa Johansson, and Robin Touche. Proving type class laws for haskell. In David van Horn and John Hughes, editors, *Trends in Functional Programming*, pages 61–74, Cham, 2019. Springer International Publishing.
- [AK16] Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: a functional datalog. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 214–227. ACM, 2016.
- [AK22] Ines Akaichi and Sabrina Kirrane. Usage control specification, enforcement, and robustness: A survey. *CoRR*, abs/2203.04800, 2022. Available at <https://arxiv.org/abs/2203.04800>.
- [AKAA<sup>+</sup>24] Jamila Alsayed Kassem, Corinne Allaart, Saba Amiri, Milen Kebede, Tim Müller, Rosanne Turner, Adam Belloum, L. Thomas van Binsbergen, Peter Grunwald, Aart van Halteren, Paola Grosso, Cees de Laat, and Sander Klous. Building a Digital Health Twin for Personalized Intervention: The EPI Project. In Boudewijn R. Haverkort, Aldert de Jongste, Pieter van Kuilenburg, and Ruben D. Vromans, editors, *Commit2Data*, volume 124 of *Open Access Series in Informatics (OASICs)*, pages 2:1–2:18, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

- [AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004.
- [Ala11] Mansaf Alam. Access specifiers model for data security in object oriented databases. In Shahram Latifi, editor, *Eighth International Conference on Information Technology: New Generations, ITNG 2011, Las Vegas, Nevada, USA, 11-13 April 2011*, pages 1068–1069. IEEE Computer Society, 2011.
- [Ams88] Jonathan Amsterdam. Some philosophical problems with formal learning theory. In Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith, editors, *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988*, pages 580–584. AAAI Press / The MIT Press, 1988.
- [ANP<sup>+</sup>03] Anne Anderson, Anthony Nadalin, B Parducci, D Engovatov, H Lockhart, M Kudo, P Humenn, S Godik, S Anderson, S Crocker, et al. extensible access control markup language (xacml) version 1.0. *Oasis*, 2003.
- [Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci.*, 14(3):329–366, 2004.
- [Arb11] Farhad Arbab. Puff, the magic protocol. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science*, pages 169–206. Springer, 2011.
- [Arb16] Farhad Arbab. Proper protocol. In Erika Ábrahám, Marcello M. Bonsangue, and Einar Broch Johnsen, editors, *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *Lecture Notes in Computer Science*, pages 65–87. Springer, 2016.
- [AS15] Yeslam Al-Saggaf. The use of data mining by private health insurance companies and customers’ privacy: an ethical analysis. *Cambridge Quarterly of Healthcare Ethics*, 24(3):281–292, 2015.
- [ASB15] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by delta-mutation. In Ahmed Bouajjani and Hugues Fauconnier, editors, *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, volume 9466 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2015.
- [AVDM05] Huib Aldewereld, Javier Vázquez-Salceda, Frank Dignum, and John-Jules Ch. Meyer. Verifying norm compliancy of protocols. In Olivier Boissier, Julian A. Padget, Virginia Dignum, Gabriela Lindemann, Eric T. Matson, Sascha Ossowski, Jaime Simão Sichman, and Javier Vázquez-Salceda, editors, *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems, AAMAS 2005 International Workshops on Agents, Norms and Institutions for Regulated Multi-Agent Systems, ANIREM 2005, and Organizations in Multi-Agent Systems, OOP 2005, Utrecht, The Netherlands, July 25-26, 2005, Revised Selected Papers*, volume 3913 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2005.
- [AvE82] Krzysztof R. Apt and Maarten H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982.
- [BBF23] Marcello Balduccini, Michael Barborak, and David A. Ferrucci. Pushing the limits of clingo’s incremental grounding and solving capabilities in practical applications. *Algorithms*, 16(3):169, 2023.

- [BBHL94] Barry W. Boehm, Prasanta K. Bose, Ellis Horowitz, and Ming June Lee. Software requirements as negotiated win conditions. In *ICRE*, pages 74–83. IEEE Computer Society, 1994.
- [BBL<sup>+</sup>17] Joshua Baugh, Ute Bartels, James Leach, Blaise Jones, Brooklyn Chaney, Katherine E Warren, Jenavieve Kirkendall, Renee Doughman, Cynthia Hawkins, Lili Miles, et al. The international diffuse intrinsic pontine glioma registry: an infrastructure to accelerate collaborative research for an orphan disease. *Journal of neuro-oncology*, 132(2):323–331, 2017.
- [BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq’Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.
- [BCFH08] Rena Bakhshi, Lucia Cloth, Wan J. Fokkink, and Boudewijn R. Haverkort. Mean-field analysis for the evaluation of gossip protocols. *SIGMETRICS Perform. Evaluation Rev.*, 36(3):31–39, 2008.
- [BF18] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.
- [BFC19] João Barbosa, Mário Florido, and Vítor Santos Costa. A three-valued semantics for typed logic programming. In Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Incelezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang, editors, *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019*, volume 306 of *EPTCS*, pages 36–51, 2019.
- [BFH<sup>+</sup>23] Alexander Bernauer, Sofia Faro, Rémy Hämmerle, Martin Huschenbett, Moritz Kiefer, Andreas Lochbihler, Jussi Mäki, Francesco Mazzoli, Simon Meier, Neil Mitchell, et al. Daml: A smart contract language for securely automating real-world multi-party business workflows. *arXiv preprint arXiv:2303.03749*, 2023. Available at <https://arxiv.org/abs/2303.03749>.
- [BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *1996 IEEE Symposium on Security and Privacy, May 6-8, 1996, Oakland, CA, USA*, pages 164–173. IEEE Computer Society, 1996.
- [BG92] Gérard Berry and Georges Gonthier. The estereel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.
- [Bia15] Pierfrancesco Biasetti. Hohfeldian normative systems. *Philosophia*, 43:951–959, 2015.
- [BILB18] Gaetano Bonfiglio, Veronica Iovinella, Gabriele Lospoto, and Giuseppe Di Battista. Kathará: A container-based framework for implementing network function virtualization and software defined networks. In *2018 IEEE/IFIP Network Operations and Management Symposium, NOMS 2018, Taipei, Taiwan, April 23-27, 2018*, pages 1–9. IEEE, 2018.
- [BL08] Ji-Won Byun and Ninghui Li. Purpose based access control for privacy protection in relational database systems. *VLDB J.*, 17(4):603–619, 2008.

- [BLGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of computer programming*, 16(2):103–149, 1991.
- [BMS08] Stefano Bistarelli, Fabio Martinelli, and Francesco Santini. Weighted datalog and levels of trust. In *Proceedings of the The Third International Conference on Availability, Reliability and Security, ARES 2008, March 4-7, 2008, Technical University of Catalonia, Barcelona , Spain*, pages 1128–1134. IEEE Computer Society, 2008.
- [BPP<sup>+</sup>19] Evmorfia Biliri, Minas Pertselakis, Marios Phinikettos, Marios Zacharias, Fenareti Lampathaki, and Dimitrios Alexandrou. Designing a trusted data brokerage framework in the aviation domain. In *Collaborative Networks and Digital Transformation: 20th IFIP WG 5.5 Working Conference on Virtual Enterprises, PRO-VE 2019, Turin, Italy, September 23–25, 2019, Proceedings 20*, pages 234–241. Springer, 2019.
- [Bro18] Simon Brown. The c4 model for visualising software architecture, 2018.
- [BS20] Gérard Berry and Manuel Serrano. Hiphop. js:(a) synchronous reactive web programming. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 533–545, 2020.
- [BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.
- [BvDdB<sup>+</sup>22] Roos Bakker, Romy AN van Drie, Maaïke de Boer, Robert van Doesburg, and Tom van Engers. Semantic role labelling for dutch law texts. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 448–457, 2022.
- [BvdT08] Guido Boella and Leendert W. N. van der Torre. Substantive and procedural norms in normative multiagent systems. *J. Appl. Log.*, 6(2):152–171, 2008.
- [BY08] Andi Bejleri and Nobuko Yoshida. Synchronous multiparty session types. In Vasco T. Vasconcelos and Nobuko Yoshida, editors, *Proceedings of the First Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@DisCoTec 2008, Oslo, Norway, June 7, 2008*, volume 241 of *Electronic Notes in Theoretical Computer Science*, pages 3–33. Elsevier, 2008.
- [BZ19] Chaouki Boulekdam and Nacer Eddine Zarour. A novel negotiation approach for requirements engineering in a cooperative context. *Multiagent Grid Syst.*, 15(3):197–218, 2019.
- [CAA<sup>+</sup>19] Z. Berkay Celik, Abbas Acar, Hidayet Aksu, Ryan Sheatsley, Patrick D. McDaniel, and A. Selcuk Uluagac. Curie: Policy-based secure data exchange. In Gail-Joon Ahn, Bhavani Thuraisingham, Murat Kantarcioglu, and Ram Krishnan, editors, *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX, USA, March 25-27, 2019*, pages 121–132. ACM, 2019.
- [Can24] Canton Team. Canton network: A network of networks for smart contract applications. Available at <https://www.digitalasset.com/hubfs/Canton/Canton%20Network%20-%20White%20Paper.pdf> (Accessed: 23-2-2024), 2024. (Whitepaper).

- [CBT22] Ruofei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite: A judgmental embedding of session types in rust. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPICs*, pages 22:1–22:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- [CCD<sup>+</sup>14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.
- [CDE<sup>+</sup>03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2003.
- [CEJP22] Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. API generation for multiparty session types, revisited and revised using scala 3. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPICs*, pages 27:1–27:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- [CFSV95] Thomas E. Cheatham, Amr F. Fahmy, Dan C. Stefanescu, and Leslie G. Valiant. Bulk synchronous parallel computing—a paradigm for transportable software. In *28th Annual Hawaii International Conference on System Sciences (HICSS-28), January 3-6, 1995, Kihei, Maui, Hawaii, USA*, pages 268–275. IEEE Computer Society, 1995.
- [CGH94] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994*, pages 213–218. Springer, 1994.
- [CGMR16] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artif. Intell.*, 231:151–181, 2016.
- [CGMT15] Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Checking termination of datalog with function symbols through linear constraints. In Domenico Lembo, Riccardo Torlone, and Andrea Marrella, editors, *23rd Italian Symposium on Advanced Database Systems, SEBD 2015, Gaeta, Italy, June 14-17, 2015*, pages 192–199. Curran Associates, Inc., 2015.
- [CGST15] Marco Calautti, Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. Checking termination of bottom-up evaluation of logic programs with function symbols. *Theory Pract. Log. Program.*, 15(6):854–889, 2015.
- [CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989.

- [Cho95] Jan Chomicki. Depth-bounded bottom-up evaluation of logic programs. *The Journal of Logic Programming*, 25(1):1–31, 1995.
- [CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.
- [Cla77] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pages 293–322, New York, 1977. Plenum Press.
- [CT22] Edward Curry and Tuomo Tuikka. An organizational maturity model for data spaces: A data sharing wheel approach. In Edward Curry, Simon Scerri, and Tuomo Tuikka, editors, *Data Spaces - Design, Deployment and Future Directions*, pages 21–42. Springer, 2022.
- [DA18] Kasper Dokter and Farhad Arbab. Reo: Textual syntax for reo connectors. In Simon Bliudze and Saddek Bensalem, editors, *Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design, MeTRiD@ETAPS 2018, Thessaloniki, Greece, 15th April 2018*, volume 272 of *EPTCS*, pages 121–135, 2018.
- [Dd16] Luís Duarte d’Almeida. Fundamental legal concepts: the hohfeldian framework. *Philosophy Compass*, 11(10):554–569, 2016.
- [dG22] Florine de Geus. Model checking normative systems. *University of Amsterdam*, 2022. Available at <https://staff.science.uva.nl/a.ponse/DeGeus.pdf>.
- [DHR01] Tyson Dowd, Fergus Henderson, and Peter Ross. Compiling mercury to the .net common language runtime. In Nick Benton and Andrew Kennedy, editors, *First International Workshop on Multi-Language Infrastructure and Interoperability, BABEL 2001, Satellite Event of PLI 2001, Firenze, Italy, September 8, 2001*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 73–88. Elsevier, 2001.
- [DLA<sup>+</sup>18] Kasper Dokter, Benjamin Lion, Farhad Arbab, Maarten Smeyers, Ali Mirlou, and Christopher A. Esterhuyse. Reo Language Compiler, 2018. Available at <https://github.com/ReoLanguage/Reo>.
- [DMGD23] Thi Van Thao Doan, Mohamed-Lamine Messai, Gérald Gavin, and Jérôme Dar-mont. A survey on implementations of homomorphic encryption schemes. *J. Supercomput.*, 79(13):15098–15139, 2023.
- [DMRT14] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. Circuits for datalog provenance. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 201–212. OpenProceedings.org, 2014.
- [dORV<sup>+</sup>22] Marcela Tuler de Oliveira, Lúcio Henrik A. Reis, Yiannis Verginadis, Diogo Menezes Ferrazani Mattos, and Sílvia Delgado Olabarriaga. Smartaccess: Attribute-based access control system for medical records based on smart contracts. *IEEE Access*, 10:117836–117854, 2022.
- [dRRdQGR<sup>+</sup>15] Rodrigo da Rosa Righi, Roberto de Quadros Gomes, Vinicius Facco Rodrigues, Cristiano André da Costa, and Antônio Marcos Alberti. Migbsp++: Improving process rescheduling on bulk-synchronous parallel applications. In *12th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2015*,

*Marrakech, Morocco, November 17-20, 2015*, pages 1–8. IEEE Computer Society, 2015.

- [dSD94] Danny de Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *J. Log. Program.*, 19/20:199–260, 1994.
- [Dun95] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.
- [Edw02a] Stephen Edwards. ESUIF: an open estereel compiler. In Florence Maraninchi, Alain Girault, and Éric Rutten, editors, *Synchronous Languages, Applications, and Programming, SLAP 2002, Satellite Event of ETAPS 2002, Grenoble, France, April 13, 2002*, volume 65 of *Electronic Notes in Theoretical Computer Science*, page 79. Elsevier, 2002.
- [Edw02b] Stephen A. Edwards. An estereel compiler for large control-dominated systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 21(2):169–183, 2002.
- [EH19] Christopher A. Esterhuyse and Hans-Dieter A. Hiep. Reowolf: Synchronous multi-party communication over the internet. In Farhad Arbab and Sung-Shik Jongmans, editors, *Formal Aspects of Component Software - 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23-25, 2019, Proceedings*, volume 12018 of *Lecture Notes in Computer Science*, pages 235–242. Springer, 2019.
- [EH24] Christopher A. Esterhuyse and Hans-Dieter A. Hiep. Reowolf 1.0 project deliverables. Record on Zenodo, 2024. Available at <https://zenodo.org/records/10838450>.
- [ELH25] Christopher Arno Esterhuyse, Benjamin Lion, and Hans-Dieter Hiep. Reowolf formalism, March 2025. Available at <https://doi.org/10.5281/zenodo.15000474>.
- [ELHA25] Christopher A. Esterhuyse, Benjamin Lion, Hans-Dieter A. Hiep, and Farhad Arbab. Formal foundations for reowolf: Multi-party sessions via synchronous protocol programming. In *COORDINATION*, volume 15731 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2025.
- [EMvB24] Christopher A. Esterhuyse, Tim Müller, and L. Thomas van Binsbergen. Justact: Actions universally justified by partial dynamic policies. In Valentina Castiglioni and Adrian Francalanza, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 44th IFIP WG 6.1 International Conference, FORTE 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14678 of *Lecture Notes in Computer Science*, pages 60–81. Springer, 2024.
- [EMvB25a] Christopher A. Esterhuyse, Tim Müller, and L. Thomas van Binsbergen. Justact+: Justified and accountable actions in policy-regulated, multi-domain data processing. *CoRR*, abs/2502.00138, 2025. Available at <https://doi.org/10.48550/arXiv.2502.00138>.
- [EMvB25b] Christopher A. Esterhuyse, Tim Müller, and L. Thomas van Binsbergen. A stable model semantics for eflint norm specifications and model checking scenarios. In *GPCE*, pages 80–93. ACM, 2025.
- [EMvBB22] Christopher A. Esterhuyse, Tim Müller, L. Thomas van Binsbergen, and Adam S. Z. Belloum. Exploring the enforcement of private, dynamic policies on medical

- workflow execution. In *18th IEEE International Conference on e-Science, e-Science 2022, Salt Lake City, UT, USA, October 11-14, 2022*, pages 481–486. IEEE, 2022.
- [End17] Mica R Endsley. From here to autonomy: lessons learned from human–automation research. *Human factors*, 59(1):5–27, 2017.
- [Est19] Christopher Esterhuysen. Safety and performance in generated coordination code. Master’s thesis, Vrije University, Amsterdam, 2019. Supervised by Farhad Arabab and Jörg Endrullis. Internship at Centrum Wiskunde & Informatica (CWI). Available at <https://ir.cwi.nl/pub/30814/30814.pdf>.
- [Est25] Christopher Arno Esterhuysen. Seaso interpreter, April 2025. Available at <https://doi.org/10.5281/zenodo.15302471>.
- [Eur16] European Commission. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance), 2016.
- [Eur20] European Commission. A european strategy for data. <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52020DC0066>, February 2020. COM(2020) 66 final.
- [EvB24] Christopher A. Esterhuysen and L. Thomas van Binsbergen. Cooperative specification via composition control. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*, pages 2–15, 2024.
- [EZ07] Stephen A. Edwards and Jia Zeng. Code generation in the columbia esterel compiler. *EURASIP J. Embed. Syst.*, 2007, 2007.
- [Fer23] Raul Castro Fernandez. Data-sharing markets: Model, protocol, and algorithms to incentivize the formation of data-sharing consortia. *Proc. ACM Manag. Data*, 1(2):172:1–172:25, 2023.
- [FJT22] Georgios Fragkos, Jay Johnson, and Eirini-Eleni Tsiropoulou. Dynamic role-based access control policy for smart grid applications: An offline deep reinforcement learning approach. *IEEE Trans. Hum. Mach. Syst.*, 52(4):761–773, 2022.
- [FYTF19] Spencer P. Florence, Shu-Hung You, Jesse A. Tov, and Robert Bruce Findler. A calculus for esterel: if can, can. if no can, no can. *Proc. ACM Program. Lang.*, 3(POPL):61:1–61:29, 2019.
- [GHK<sup>+</sup>15] Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. Abstract gringo. *Theory and Practice of Logic Programming*, 15(4-5):449–463, 2015.
- [GIM<sup>+</sup>18] Guido Governatori, Florian Idelberger, Zoran Milosevic, Régis Riveret, Giovanni Sartor, and Xiwei Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artif. Intell. Law*, 26(4):377–409, 2018.
- [GKK<sup>+</sup>11] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.
- [GKKS19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019.

- [GKKS22] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer set solving in practice*. Springer Nature, 2022.
- [GKS12] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Multi-threaded ASP solving with clasp. *Theory Pract. Log. Program.*, 12(4-5):525–545, 2012.
- [GKV<sup>+</sup>20] Casandra Grundstrom, Olli Korhonen, Karin Väyrynen, Minna Isomursu, et al. Insurance customers’ expectations for sharing health data: qualitative survey study. *JMIR medical informatics*, 8(3):e16102, 2020.
- [GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.
- [GLGB87] Thierry Gautier, Paul Le Guernic, and Loic Besnard. *Signal: A declarative language for synchronous programming of real-time systems*. Springer, 1987.
- [GM84] Joseph A. Goguen and José Meseguer. Equality, types, modules, and (why not ?) generics for logic programming. *J. Log. Program.*, 1(2):179–210, 1984.
- [GMS06] Guido Governatori, Zoran Milosevic, and Shazia Wasim Sadiq. Compliance checking between business processes and business contracts. In *Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16-20 October 2006, Hong Kong, China*, pages 221–232. IEEE Computer Society, 2006.
- [GMT13] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments. *Theory Pract. Log. Program.*, 13(4-5):737–752, 2013.
- [GN25] Vinod Gurav and Sudhir R Nagarkar. Zenodo: A platform for open access and sustainable digital research repository. *TechnoLibrarianship: A Gateway Towards Future Libraries-2025, Shivaji University, Kolhapur, PP-152-159*, 2025.
- [GRJ22] Katharina Großer, Volker Riediger, and Jan Jürjens. Requirements document relations: A reuse perspective on traceability through standards. *Software and Systems Modeling*, 21(6):1–37, 2022.
- [Gro11] William Gropp. MPI (message passing interface). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1184–1190. Springer, 2011.
- [GV94] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel Distributed Comput.*, 22(2):251–267, 1994.
- [Hal05] Nicolas Halbwachs. A synchronous language at work: the story of lustre. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11-14 July 2005, Verona, Italy, Proceedings*, pages 3–11. IEEE Computer Society, 2005.
- [Han94] Sven Ove Hansson. Review of deontic logic in computer science: Normative system specification, john-jules ch. meyer and roel j. wieringa (eds.), john wiley & sons, chichester 1993. *Bull. IGPL*, 2(2):249–251, 1994.
- [Han13] Michael Hanus. Functional logic programming: From theory to curry. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Computer Science*, pages 123–168. Springer, 2013.

- [HCP22] Giray Havur, Cristina Cabanillas, and Axel Polleres. Benchmarking answer set programming systems for resource allocation in business processes. *Expert Syst. Appl.*, 205:117599, 2022.
- [HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, 1991.
- [He06] Yujing He. Comparison of the modeling languages alloy and UML. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*, pages 671–677. CSREA Press, 2006.
- [HHB14] Fei Hu, Qi Hao, and Ke Bao. A survey on software-defined network and openflow: From concept to implementation. *IEEE Commun. Surv. Tutorials*, 16(4):2181–2206, 2014.
- [HIA+23] Ali Hariri, Amjad Ibrahim, Bithin Alangot, Subhajit Bandopadhyay, Antonio La Marra, Alessandro Rosetti, Hussein Joumaa, and Theo Dimitrakos. *UCON+: Comprehensive Model, Architecture and Implementation for Usage Control and Continuous Authorization*, pages 209–226. Springer International Publishing, Cham, 2023.
- [HKF15] Vincent C. Hu, D. Richard Kuhn, and David F. Ferraiolo. Attribute-based access control. *Computer*, 48(2):85–88, 2015.
- [HLY13] Amelia Harrison, Vladimir Lifschitz, and Fangkai Yang. On the semantics of gringo. *arXiv preprint arXiv:1312.6149*, 2013.
- [Hon93] Kohei Honda. Types for dyadic interaction. In *International Conference on Concurrency Theory*, pages 509–523. Springer, 1993.
- [Hor51] Alfred Horn. On sentences which are true of direct unions of algebras. *J. Symb. Log.*, 16(1):14–21, 1951.
- [HPS21] Stijn Henckens, Mostafa Mohajeri Parizi, and Giovanni Sileno. Enriching a cp-net by asymmetric merging. *CoRR*, abs/2109.10110, 2021. Available at <https://arxiv.org/abs/2109.10110>.
- [HR00] Mark A Hall and Stephen S Rich. Genetic privacy laws and patients’ fear of discrimination by health insurers: The view from genetic counselors. *Journal of Law, Medicine & Ethics*, 28(3):245–257, 2000.
- [HW10] Adrian Hornsby and Rod Walsh. From instant messaging to cloud computing, an xmpp review. In *IEEE international symposium on consumer electronics (ISCE 2010)*, pages 1–6. IEEE, 2010.
- [HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008.
- [Ian07] Renato Ianella. Open digital rights language (odrl). *Open Content Licensing: Cultivating the Creative Commons*, 2007.
- [ILN22] Keigo Imai, Julien Lange, and Rumyana Neykova. Kmclib: Automated inference and verification of session types from ocaml programs. In Dana Fisman and

Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 379–386. Springer, 2022.

- [JA12] Sung-Shik TQ Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for reo. *Scientific Annals of Computer Science*, 22(1), 2012.
- [JA15] Sung-Shik T. Q. Jongmans and Farhad Arbab. Can high throughput atone for high latency in compiler-generated protocol code? In Mehdi Dastani and Marjan Sirjani, editors, *Fundamentals of Software Engineering - 6th International Conference, FSEN 2015 Tehran, Iran, April 22-24, 2015, Revised Selected Papers*, volume 9392 of *Lecture Notes in Computer Science*, pages 238–258. Springer, 2015.
- [Jac03] Daniel Jackson. Alloy: A logical modelling language. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*, page 1. Springer, 2003.
- [Jac19] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019.
- [JD22] Christian Jung and Jörg Dörr. *Data Usage Control*, pages 129–146. Springer International Publishing, Cham, 2022.
- [JJA12] Johan Jeuring, Patrik Jansson, and Cláudio Amaral. Testing type class laws. *SIGPLAN Not.*, 47(12):49–60, sep 2012.
- [JSS+12] Sung-Shik T. Q. Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. Automatic code generation for the orchestration of web services with reo. In Flavio De Paoli, Ernesto Pimentel, and Gianluigi Zavattaro, editors, *Service-Oriented and Cloud Computing - First European Conference, ESOC 2012, Bertinoro, Italy, September 19-21, 2012. Proceedings*, volume 7592 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.
- [JSS+14] Sung-Shik T. Q. Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. Orchestrating web services using reo: from circuits and behaviors to automatically generated code. *Serv. Oriented Comput. Appl.*, 8(4):277–297, 2014.
- [JSS16] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 422–430. Springer, 2016.
- [KAA+24] Jamila Alsayed Kassem, Corinne G. Allaart, Saba Amiri, Milen G. Kebede, Tim Müller, Rosanne Turner, Adam Belloum, L. Thomas van Binsbergen, Peter Grunwald, Aart van Halteren, Paola Grosso, Cees de Laat, and Sander Klous. Building a digital health twin for personalized intervention: The EPI project. In Boudewijn R. Haverkort, Aldert de Jongste, Pieter van Kuilenburg, and Ruben D. Vromans, editors, *Commit2Data*, volume 124 of *OASICs*, pages 2:1–2:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

- [KCPA08] Christian Krause, David Costa, José Proença, and Farhad Arbab. Reconfiguration of reo connectors triggered by dataflow. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 10, 2008.
- [KdHH22] Misbah Khan, Frank T. H. den Hartog, and Jiankun Hu. A survey and ontology of blockchain consensus algorithms for resource-constrained iot systems. *Sensors*, 22(21):8188, 2022.
- [KdLTG20] Jamila Alsayed Kassem, Cees de Laat, Arie Taal, and Paola Grosso. The EPI framework: A dynamic data sharing framework for healthcare use cases. *IEEE Access*, 8:179909–179920, 2020.
- [KGV13] C. Krause, H. Giese, and E.P. Vink, de. Compositional and behavior-preserving reconfiguration of component connectors in reo. *Journal of Visual Languages and Computing*, 24(3):153–168, 2013.
- [KK20] Bas Ketsman and Christoph Koch. Datalog with negation and monotonicity. In Carsten Lutz and Jean Christoph Jung, editors, *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*, volume 155 of *LIPICs*, pages 19:1–19:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [KK22] Bas Ketsman and Paraschos Koutris. Modern datalog engines. *Found. Trends Databases*, 12(1):1–68, 2022.
- [KKS21] Fabian Kneer, Erik Kamsties, and Klaus Schmid. Adaptationexplore - A process for elicitation, negotiation, and documentation of adaptive requirements. In *REFSQ*, volume 12685 of *Lecture Notes in Computer Science*, pages 81–98. Springer, 2021.
- [KME<sup>+</sup>24] Jamila Alsayed Kassem, Tim Müller, Christopher A. Esterhuyse, Milen G. Kebede, Anwar Osseyran, and Paola Grosso. The epi framework: A data privacy by design framework to support healthcare use cases. *Future Generation Computer Systems*, page 107550, 2024.
- [KMLA11] Christian Krause, Ziyang Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in reo using high-level replacement systems. *Sci. Comput. Program.*, 76(1):23–36, 2011.
- [KMT12] Adrian Kuhn, Gail C. Murphy, and C. Albert Thompson. An exploratory study of forces and frictions affecting large-scale model-driven development. In Robert B. France, Jürgen Kazmeier, Ruth Brey, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2012.
- [KPE24] David Klopp, André Pacak, and Sebastian Erdweg. Separate compilation and partial linking: Modules for datalog ir. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 94–106, 2024.
- [Kra11] Christian Krause. *Reconfigurable Component Connectors*. PhD thesis, Leiden University, 2011.
- [Kra18] Tim Kraska. Northstar: An interactive data science system. *Proc. VLDB Endow.*, 11(12):2150–2164, 2018.
- [KRDO17] Aggelos Kiyayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan

- Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.
- [KS94] Robert A. Kowalski and Fariba Sadri. The situation calculus and event calculus compared. In Maurice Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994*, pages 539–553. MIT Press, 1994.
- [KSKP24] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. Coqilot, a plugin for llm-based generation of proofs. In *ASE*, pages 2382–2385. ACM, 2024.
- [KT22] Pooja Khobragade and Ashok Kumar Turuk. Blockchain consensus algorithms: A survey. In Javier Prieto, Francisco Luis Benítez Martínez, Stefano Ferretti, David Arroyo Guardado, and Pedro Tomás Nevado-Batalla, editors, *Blockchain and Applications, 4th International Congress, BLOCKCHAIN 2022, L'Aquila, Italy, 13-15 July 2022*, volume 595 of *Lecture Notes in Networks and Systems*, pages 198–210. Springer, 2022.
- [KVBG21] Jamila Alsayed Kassem, Onno Valkering, Adam Belloum, and Paola Grosso. EPI framework: Approach for traffic redirection through containerised network functions. In *17th IEEE International Conference on eScience, eScience 2021, Innsbruck, Austria, September 20-23, 2021*, pages 80–89. IEEE, 2021.
- [LAT22] Benjamin Lion, Farhad Arbab, and Carolyn L. Talcott. A semantic model for interacting cyber-physical systems. *J. Log. Algebraic Methods Program.*, 129:100807, 2022.
- [LdPMS25] Diogo Landau, Ingeborg de Pater, Mihaela Mitici, and Nishant Saurabh. Federated learning framework for collaborative remaining useful life prognostics: an aircraft engine case study. *arXiv preprint arXiv:2506.00499*, 2025.
- [LGF03] Ninghui Li, Benjamin N. Grosz, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, 2003.
- [Lif10] Vladimir Lifschitz. Datalog programs and their stable models. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 78–87. Springer, 2010.
- [LKR13] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. Network innovation using openflow: A survey. *IEEE communications surveys & tutorials*, 16(1):493–512, 2013.
- [Llo84] John W. Lloyd. *Foundations of Logic Programming, 1st Edition*. Springer, 1984.
- [LM03] Ninghui Li and John C. Mitchell. DATALOG with constraints: A foundation for trust management languages. In Verónica Dahl and Philip Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA, January 13-14, 2003, Proceedings*, volume 2562 of *Lecture Notes in Computer Science*, pages 58–73. Springer, 2003.
- [LM16] Sam Lindley and J. Garrett Morris. Embedding session types in haskell. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on*

*Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 133–145. ACM, 2016.

- [LSvE20] Lu-Chi Liu, Giovanni Sileno, and Tom M. van Engers. Digital enforceable contracts (DEC): making smart contracts smarter. In Serena Villata, Jakub Harasta, and Petr Kremen, editors, *Legal Knowledge and Information Systems - JURIX 2020: The Thirty-third Annual Conference, Brno, Czech Republic, December 9-11, 2020*, volume 334 of *Frontiers in Artificial Intelligence and Applications*, pages 235–238. IOS Press, 2020.
- [LTK<sup>+</sup>18] Grischa Liebel, Matthias Tichy, Eric Knauss, Oscar Ljungkrantz, and Gerald Stieglbauer. Organisation and communication problems in automotive requirements engineering. *Requirements Engineering*, 23:145–167, 2018.
- [LZZ<sup>+</sup>19] Bing Lin, Fangning Zhu, Jianshan Zhang, Jiaqing Chen, Xing Chen, Neal Naixue Xiong, and Jaime Lloret Mauri. A time-driven data placement strategy for a scientific workflow combining edge computing and cloud computing. *IEEE Trans. Ind. Informatics*, 15(7):4254–4265, 2019.
- [MAB<sup>+</sup>08] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. Openflow: enabling innovation in campus networks. *Comput. Commun. Rev.*, 38(2):69–74, 2008.
- [McC63] John McCarthy. Situations, actions, and causal laws. (*No Title*), 1963.
- [McC02] John McCarthy. Actions and other events in situation calculus. In *Kr*, pages 615–628, 2002.
- [MEvB25] Tim Müller, Christopher A. Esterhuysen, and L. Thomas van Binsbergen. Source code and experimental data for a translation from eflint normative specifications to clingo answer-set programs. Record on Zenodo, 2025. Available at <https://zenodo.org/records/15188959>.
- [MK20] Konstantinos Mokoš and Panagiotis Katsaros. A survey on the formalisation of system requirements and their validation. *Array*, 7:100030, 2020.
- [ML08] Sanjay Modgil and Michael Luck. Argumentation based resolution of conflicts between desires and normative goals. In Iyad Rahwan and Pavlos Moraitis, editors, *Argumentation in Multi-Agent Systems, Fifth International Workshop, ArgMAS 2008, Estoril, Portugal, May 12, 2008. Revised Selected and Invited Papers*, volume 5384 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2008.
- [MLP<sup>+</sup>19] Andres Munoz-Arcentales, Sonsoles López-Pernas, Alejandro Pozo, Álvaro Alonso, Joaquín Salvachúa, and Gabriel Huecas. An architecture for providing data usage and access control in data sharing ecosystems. In Elhadi M. Shakshuki, Ansar-Ul-Haque Yasar, and Haroon Malik, editors, *The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2019) / The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2019) / Affiliated Workshops, Coimbra, Portugal, November 4-7, 2019*, volume 160 of *Procedia Computer Science*, pages 590–597. Elsevier, 2019.
- [MP11] Sanjay Modgil and Henry Prakken. Revisiting preferences and argumentation. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 1021–1026. IJCAI/AAAI, 2011.

- [MP15] Rodrigo M. L. M. Moreira and Ana C. R. Paiva. A novel approach using alloy in domain-specific language engineering. In Slimane Hammoudi, Luís Ferreira Pires, Philippe Desfray, and Joaquim Filipe, editors, *MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, Angers, Loire Valley, France, 9-11 February, 2015*, pages 157–164. SciTePress, 2015.
- [MTKW18] David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. Datalog: concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, volume 20 of *ACM Books*, pages 3–100. ACM / Morgan & Claypool, 2018.
- [MWY17] Xinjun Mao, QiuZhen Wang, and Sen Yang. A survey of agent-oriented programming from software engineering perspective. *Web Intell.*, 15(2):143–163, 2017.
- [MZ17] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.
- [Ney13] Rumyana Neykova. Session types go dynamic or how to verify your python conversations. In Nobuko Yoshida and Wim Vanderbauwhede, editors, *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013*, volume 137 of *EPTCS*, pages 95–102, 2013.
- [NO18] Juan Carlos Nieves and Mauricio Osorio. Extending well-founded semantics with clark’s completion for disjunctive logic programs. *Sci. Program.*, 2018:4157030:1–4157030:10, 2018.
- [OER23] Roy Overbeek, Jörg Endrullis, and Aloïs Rosset. Graph rewriting and relabeling with pbpo<sup>+</sup>: A unifying theory for quasitoposes. *J. Log. Algebraic Methods Program.*, 133:100873, 2023.
- [Olk16a] Grigory K. Olkhovikov. A complete, correct, and independent axiomatization of the first-order fragment of a three-valued paraconsistent logic. *FLAP*, 3(3):335–340, 2016.
- [Olk16b] Grigory K. Olkhovikov. On a new three-valued paraconsistent logic. *FLAP*, 3(3):317–334, 2016.
- [OSTL19] B. Otto, S. Steinbuss, A. Teuscher, and S Lohmann. Ids reference architecture model – version 3.0, 4 2019.
- [OtHW22] Boris Otto, Michael ten Hompel, and Stefan Wrobel, editors. *Designing Data Spaces: The Ecosystem Approach to Competitive Advantage*. Springer, 2022.
- [Ott22] Boris Otto. The evolution of data spaces. In Boris Otto, Michael ten Hompel, and Stefan Wrobel, editors, *Designing Data Spaces: The Ecosystem Approach to Competitive Advantage*, pages 3–15. Springer, 2022.
- [OZ19] Mauricio Osorio and Claudia Zepeda. Three new genuine five-valued logics intended to model non-trivial concepts. In Pilar Pozos Parra and José Raymundo Marcial-Romero, editors, *Selected Papers of the Eleventh and Twelfth Latin American Workshop on Logic/Languages, Algorithms and New Methods of Reasoning, LANMR 2018, Puebla, Mexico, November 15, 2018 & LANMR 2019*,

Puebla, Mexico, November 15, 2019, volume 354 of *Electronic Notes in Theoretical Computer Science*, pages 157–170. Elsevier, 2019.

- [Pac24] André Pacak. *Datalog as a (non-logic) Programming Language*. PhD thesis, University of Mainz, Germany, 2024.
- [PBEB07] Dumitru Potop-Butucaru, Stephen A Edwards, and Gérard Berry. *Compiling estereel*, volume 86. Springer Science & Business Media, 2007.
- [PC20] José Proença and Guillermina Cledou. Arx: reactive programming for synchronous connectors. In *Coordination Models and Languages: 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valtta, Malta, June 15–19, 2020, Proceedings 22*, pages 39–56. Springer, 2020.
- [PCdVA12] José Proença, Dave Clarke, Erik P. de Vink, and Farhad Arbab. Dreams: a framework for distributed synchronous coordination. In Sascha Ossowski and Paola Lecca, editors, *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26–30, 2012*, pages 1510–1515. ACM, 2012.
- [PCG<sup>+</sup>23a] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. University of Pennsylvania, 2023. Version 6.4.
- [PCG<sup>+</sup>23b] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. University of Pennsylvania, 2023. Version 6.4.
- [Pea94] Judea Pearl. Causation, action and counterfactuals. In Anthony G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence, Amsterdam, The Netherlands, August 8–12, 1994*, pages 826–828. John Wiley and Sons, Chichester, 1994.
- [PHB06] Alexander Pretschner, Manuel Hilty, and David Basin. Distributed usage control. *Commun. ACM*, 49(9):39–44, sep 2006.
- [PM12] Henry Prakken and Sanjay Modgil. Clarifying some misconceptions on the aspic<sup>+</sup> framework. In Bart Verheij, Stefan Szeider, and Stefan Woltran, editors, *Computational Models of Argument - Proceedings of COMMA 2012, Vienna, Austria, September 10–12, 2012*, volume 245 of *Frontiers in Artificial Intelligence and Applications*, pages 442–453. IOS Press, 2012.
- [PRR<sup>+</sup>24] Alireza Parvizimosaed, Marco Roveri, Aidin Rasti, Amal Ahmed Anda, Sofana Alfuhaid, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Symboleop: checking properties of legal contracts. *Software and Systems Modeling*, pages 1–34, 2024.
- [Prz90] Teodor C. Przymusiński. The well-founded semantics coincides with the three-valued stable semantics. *Fundam. Inform.*, 13(4):445–463, 1990.
- [PSA<sup>+</sup>20] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Subcontracting, assignment, and substitution for legal contracts in symboleo. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling - 39th International Conference, ER 2020, Vienna, Austria, November 3–6, 2020, Proceedings*, volume 12400 of *Lecture Notes in Computer Science*, pages 271–285. Springer, 2020.

- [PSA<sup>+</sup>22] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, Marco Roveri, Aidin Rasti, Ali Roudak, and John Mylopoulos. Specification and analysis of legal contracts with symboleo. *Softw. Syst. Model.*, 21(6):2395–2427, 2022.
- [PvBSvE22] Mostafa Mohajeri Parizi, L. Thomas van Binsbergen, Giovanni Sileno, and Tom van Engers. A modular architecture for integrating normative advisors in mas. In Dorothea Baumeister and Jörg Rothe, editors, *Multi-Agent Systems*, pages 312–329, Cham, 2022. Springer International Publishing.
- [PWZ<sup>+</sup>17] Johannes Pillmann, Christian Wietfeld, Adrian Zarcula, Thomas Raugust, and Daniel Calvo Alonso. Novel common vehicle information model (CVIM) for future automotive vehicle big data marketplaces. In *Intelligent Vehicles Symposium*, pages 1910–1915. IEEE, 2017.
- [QTD<sup>+</sup>20] Jing Qiu, Zhihong Tian, Chunlai Du, Qi Zuo, Shen Su, and Binxing Fang. A survey on access control in the age of internet of things. *IEEE Internet Things J.*, 7(6):4682–4696, 2020.
- [RBA05] Wei Ren, Randal W. Beard, and Ella M. Atkins. A survey of consensus problems in multi-agent coordination. In *American Control Conference, ACC 2005, Portland, OR, USA, 8-10 June, 2005*, pages 1859–1864. IEEE, 2005.
- [REC<sup>+</sup>15] Karen Rose, Scott Eldridge, Lyman Chapin, et al. The internet of things: An overview. *The internet society (ISOC)*, 80(15):1–53, 2015.
- [RG02] Mark Richters and Martin Gogolla. OCL: syntax, semantics, and tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, pages 42–68. Springer, 2002.
- [RM22] Sandeep Reddivari and Ahmed Moussa. Renego: A requirements negotiation tool based on voting of social choice theory. In *COMPSAC*, pages 436–437. IEEE, 2022.
- [Ros90] Kenneth A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In Daniel J. Rosenkrantz and Yehoshua Sagiv, editors, *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*, pages 161–171. ACM Press, 1990.
- [RU95] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *J. Log. Program.*, 23(2):125–149, 1995.
- [S<sup>+</sup>06] Douglas C Schmidt et al. Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25, 2006.
- [SA19] Christian Strasser and G. Aldo Antonelli. Non-monotonic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2019 edition, 2019.
- [Sac11] Krzysztof Sacha. Trust management languages and complexity. In Robert Meersman, Tharam S. Dillon, Pilar Herrero, Akhil Kumar, Manfred Reichert, Li Qing, Beng Chin Ooi, Ernesto Damiani, Douglas C. Schmidt, Jules White, Manfred Hauswirth, Pascal Hitzler, and Mukesh K. Mohania, editors, *On the Move to Meaningful Internet Systems: OTM 2011 - Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17-21, 2011, Proceedings, Part II*, volume 7045 of *Lecture Notes in Computer Science*, pages 588–604. Springer, 2011.

- [San98] Ravi S. Sandhu. Role-based access control. *Adv. Comput.*, 46:237–286, 1998.
- [SBvE15] Giovanni Sileno, Alexander Boer, and Tom M. van Engers. A constructivist approach to rule bases. In Stéphane Loiseau, Joaquim Filipe, Béatrice Duval, and H. Jaap van den Herik, editors, *ICAART 2015 - Proceedings of the International Conference on Agents and Artificial Intelligence, Volume 2, Lisbon, Portugal, 10-12 January, 2015*, pages 540–547. SciTePress, 2015.
- [SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
- [SdV00] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design, Tutorial Lectures [revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design, FOSAD 2000, Bertinoro, Italy, September 2000]*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer, 2000.
- [Sha01] Murray Shanahan. The event calculus explained. In *Artificial intelligence today: Recent trends and developments*, pages 409–430. Springer, 2001.
- [SHC94] Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. The implementation of mercury, an efficient purely declarative logic programming language. In Koenraad De Bosschere, Bart Demoen, and Paul Tarau, editors, *ILPS 1994, Workshop 4: Implementation Techniques for Logic Programming Languages, Ithaca, New York, USA, November 17, 1994*, 1994.
- [SHH24] Eli Silver, Robert Hunt, and Daniel Harris. Fievel: a fast inexpensive velocimeter based on an optical mouse sensor. *Bulletin of the American Physical Society*, 2024.
- [Shi11] Kate Shires. How to protect data in times of change? a review of the eu commission’s and ico’s response to updating the law on data protection. *Journal of Database Marketing & Customer Strategy Management*, 18:65–68, 2011.
- [Sho93] Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993.
- [SK12] Birgit Schmidt and Iryna Kuchma. *Implementing open access mandates in Europe: OpenAIRE study on the development of open access repository communities in Europe*. Universitätsverlag Göttingen, 2012.
- [SM17] Dipa Soni and Ashwin Makwana. A survey on mqtt: a protocol of internet of things (iot). In *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, volume 20, pages 173–177, 2017.
- [SMHB98] Jawed I. A. Siddiqi, Ian C. Morrey, Richard Hibberd, and Graham Buckberry. Understanding and exploring formal specifications. *Ann. Softw. Eng.*, 6:411–432, 1998.
- [SMO24] Jorrit Stutterheim, Aleandro Mifsud, and Ana Oprescu. Dynamos: Dynamic microservice composition for data-exchange systems, lessons learned. In *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, pages 8–15, 2024.
- [SMV+19] Sara Shakeri, Valentina Maccatrozzo, Lourens E. Veen, Rena Bakhshi, Leon Gommans, Cees de Laat, and Paola Grosso. Modeling and matching digital data marketplace policies. In *15th International Conference on eScience, eScience 2019, San Diego, CA, USA, September 24-27, 2019*, pages 570–577. IEEE, 2019.

- [SO17] Daniel Servos and Sylvia L. Osborn. Current research and open problems in attribute-based access control. *ACM Comput. Surv.*, 49(4):65:1–65:45, 2017.
- [SP03] Ravi Sandhu and Jaehong Park. Usage control: A vision for next generation access control. In Vladimir Gorodetsky, Leonard Popyack, and Victor Skormin, editors, *Computer Network Security*, pages 17–31, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- [SPA+20] Sepehr Sharifi, Alireza Parvizmosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Symboleo: Towards a specification language for legal contracts. In Travis D. Breaux, Andrea Zisman, Samuel Fricker, and Martin Glinz, editors, *28th IEEE International Requirements Engineering Conference, RE 2020, Zurich, Switzerland, August 31 - September 4, 2020*, pages 364–369. IEEE, 2020.
- [ST06] Stephan Schiffel and Michael Thielscher. Reconciling situation calculus and fluent calculus. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 287–292. AAAI Press, 2006.
- [Suh19] Muhammad Suhaib. Conflicts identification among stakeholders in goal oriented requirements engineering process. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 2019.
- [SvBPvE22] Giovanni Sileno, Thomas van Binsbergen, Matteo Pascucci, and Tom van Engers. Dpcl: a language template for normative specifications. *arXiv preprint arXiv:2201.04477*, 2022. Available at <https://arxiv.org/abs/2201.04477>.
- [SVG20] Sara Shakeri, Lourens E. Veen, and Paola Grosso. Evaluation of container overlays for secure data sharing. In Hwee-Pink Tan, Lyes Khoukhi, and Sharief Oteafy, editors, *45th IEEE LCN Symposium on Emerging Topics in Networking, LCN Symposium 2020, Sydney, Australia, November 16-19, 2020*, pages 99–108. IEEE, 2020.
- [SZ94] Baile Shi and Aoying Zhou. Bottom-up evaluation of datalog with negation. *J. Comput. Sci. Technol.*, 9(3):229–244, 1994.
- [Sza97] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.
- [Tar41] Alfred Tarski. On the calculus of relations. *The journal of symbolic logic*, 6(3):73–89, 1941.
- [TCM24] Charlotte Tschider, Marcelo Corrales Compagnucci, and Timo Minssen. The new eu–us data protection framework’s implications for healthcare. *Journal of Law and the Biosciences*, 11(2):lsae022, 2024.
- [TGMD22] Ana I. Torre-Bastida, Guillermo Gil, Raúl Miñón, and Josu Díaz-de-Arcaya. Technological perspective of data governance in data space ecosystems. In Edward Curry, Simon Scerri, and Tuomo Tuikka, editors, *Data Spaces - Design, Deployment and Future Directions*, pages 65–87. Springer, 2022.
- [Thi00] Michael Thielscher. The fluent calculus. Technical report, Tech. Rep. CL-2000-01, Dresden University of Technology, 2000.
- [UBL+11] Andrzej Uszok, Jeffrey M. Bradshaw, James Lott, Matthew Johnson, Maggie R. Breedy, Michael Vignati, Keith Whittaker, Kimberly Jakubowski, Jeffrey Bowcock, and Daniel Apgar. Toward a flexible ontology-based policy approach for network

- operations using the kaos framework. In *MILCOM 2011 - 2011 IEEE Military Communications Conference, Baltimore, MD, USA, November 7-10, 2011*, pages 1108–1114. IEEE, 2011.
- [UMSB12] Um-e-Ghazia, Rahat Masood, Muhammad Awais Shibli, and Muhammad Bilal. Usage control model specification in XACML policy language - XACML policy engine of UCON. In Agostino Cortesi, Nabendu Chaki, Khalid Saeed, and Slawomir T. Wierzchon, editors, *Computer Information Systems and Industrial Management - 11th IFIP TC 8 International Conference, CISIM 2012, Venice, Italy, September 26-28, 2012. Proceedings*, volume 7564 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2012.
- [Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
- [vBKB<sup>+</sup>21] L. Thomas van Binsbergen, Milen G. Kebede, Joshua Baugh, Tom M. van Engers, and Dannis G. van Vuurden. Dynamic generation of access control policies from social policies. In Nuno Varandas, Ansar-Ul-Haque Yasar, Haroon Malik, and Stéphane Galland, editors, *The 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2021) / The 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2021), Leuven, Belgium, November 1-4, 2021*, volume 198 of *Procedia Computer Science*, pages 140–147. Elsevier, 2021.
- [vBLvDvE20] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom M. van Engers. eflint: a domain-specific language for executable norm specifications. In Martin Erwig and Jeff Gray, editors, *GPCE '20: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Virtual Event, USA, November 16-17, 2020*, pages 124–136. ACM, 2020.
- [vBORS<sup>+</sup>24] L. Thomas van Binsbergen, Merrick Oost-Rosengren, Hayo Schreijer, Freek Dijkstra, and Taco van Dijk. *AMdEX Reference Architecture – version 1.0.0. 2* 2024. Available at [10.5281/zenodo.10565915](https://doi.org/10.5281/zenodo.10565915).
- [VC07] Francesco Viganò and Marco Colombetti. Symbolic model checking of institutions. In Maria L. Gini, Robert J. Kauffman, Donna Sarppo, Chrysanthos Dellarocas, and Frank Dignum, editors, *Proceedings of the 9th International Conference on Electronic Commerce: The Wireless World of Electronic Commerce, 2007, University of Minnesota, Minneapolis, MN, USA, August 19-22, 2007*, volume 258 of *ACM International Conference Proceeding Series*, pages 35–44. ACM, 2007.
- [VC08] Francesco Viganò and Marco Colombetti. Model checking norms and sanctions in institutions. In Jaime Simão Sichman, Julian Padget, Sascha Ossowski, and Pablo Noriega, editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems III*, pages 316–329. Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [VCB21] Onno Valkering, Reginald Cushing, and Adam Belloum. Brane: A framework for programmable orchestration of multi-site applications. In *17th IEEE International Conference on eScience, eScience 2021, Innsbruck, Austria, September 20-23, 2021*, pages 277–282. IEEE, 2021.
- [vdA03] Wil M. P. van der Aalst. Business process management demystified: A tutorial on models, systems and standards for workflow management. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*,

*Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer, 2003.

- [vdAC22] Wil M. P. van der Aalst and Josep Carmona, editors. *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*. Springer, 2022.
- [vdAvH04] Wil van der Aalst and Kees Max van Hee. *Workflow management: models, methods, and systems*. MIT press, 2004.
- [vdTV14] Leendert W. N. van der Torre and Serena Villata. An aspic-based legal argumentation framework for deontic reasoning. In Simon Parsons, Nir Oren, Chris Reed, and Federico Cerutti, editors, *Computational Models of Argument - Proceedings of COMMA 2014, Atholl Palace Hotel, Scottish Highlands, UK, September 9-12, 2014*, volume 266 of *Frontiers in Artificial Intelligence and Applications*, pages 421–432. IOS Press, 2014.
- [vDvdSvE16] Robert van Doesburg, Tijs van der Storm, and Tom van Engers. Calculemus: towards a formal language for the interpretation of normative systems. *AI4J Artif Intell Justice*, 1:73, 2016.
- [vEK76] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.
- [VG89] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '89, pages 1–10, New York, NY, USA, March 1989. Association for Computing Machinery.
- [VGRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):619–649, July 1991.
- [VHG20] Andrea Vázquez-Ingelmo, Alicia García Holgado, and Francisco J. García-Peñalvo. C4 model in a software engineering subject to ease the comprehension of UML and the software. In *2020 IEEE Global Engineering Education Conference, EDUCON 2020, Porto, Portugal, April 27-30, 2020*, pages 919–924. IEEE, 2020.
- [Vil10] Serena Villata. A normative multiagent approach to requirements engineering. *Log. J. IGPL*, 18(1):245–274, 2010.
- [VvEI18] Laurens Versluis, Erwin van Eyk, and Alexandru Iosup. An analysis of workflow formalisms for workflows with complex non-functional requirements. In Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar, editors, *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*, pages 107–112. ACM, 2018.
- [W3C12] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview (Second Edition), 12 2012.
- [WBGs23] Evan Wang, Yogesh Barve, Aniruddha Gokhale, and Hongyang Sun. Dynamic resource management for cloud-native bulk synchronous parallel applications. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 152–157, 2023.

- [WdPAOdPL09] Vera Maria Benjamim Werneck, Antonio de Pádua Albuquerque Oliveira, and Julio Cesar Sampaio do Prado Leite. Comparing GORE frameworks: i-star and KAOS. In Claudia P. Ayala, Carla T. L. L. Silva, and Hernán Astudillo, editors, *Anais do WER09 - Workshop em Engenharia de Requisitos, Valparaíso, Chile, Julho 16-17, 2009*, 2009.
- [Wes13] Newcomb Hohfeld Wesley. Some fundamental legal conceptions as applied in judicial reasoning. *The Yale Law Journal*, 23(1):16, 1913.
- [Yu01] Eric Yu. Agent orientation as a modelling paradigm. *Wirtschaftsinformatik*, 43:123–132, 2001.
- [Yu09] Eric S. K. Yu. Social modeling and i\*. In Alexander Borgida, Vinay K. Chaudhri, Paolo Giorgini, and Eric S. K. Yu, editors, *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos*, volume 5600 of *Lecture Notes in Computer Science*, pages 99–121. Springer, 2009.
- [ZBL+23] Xin Zhou, Adam Belloum, Michael Harold Lees, Tom M. van Engers, and Cees de Laat. The dynamics of corruption under an optional external supervision service. *Appl. Math. Comput.*, 457:128172, 2023.
- [ZC08] Gansen Zhao and David W. Chadwick. On the modeling of bell-lapadula security policies using RBAC. In *17th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, WETICE 2008, Rome, Italy, June 23-25, 2008, Proceedings*, pages 257–262. IEEE Computer Society, 2008.
- [ZCG+19] Lu Zhang, Reginald Cushing, Leon Gommans, Cees T. A. M. de Laat, and Paola Grosso. Modeling of collaboration archetypes in digital market places. *IEEE Access*, 7:102689–102700, 2019.
- [ZMKvE22] Tomasz Zurek, Mostafa Mohajeriparizi, Jonathan Kwik, and Tom M. van Engers. Can a military autonomous device follow international humanitarian law? In Enrico Francesconi, Georg Borges, and Christoph Sorge, editors, *Legal Knowledge and Information Systems - JURIX 2022: The Thirty-fifth Annual Conference, Saarbrücken, Germany, 14-16 December 2022*, volume 362 of *Frontiers in Artificial Intelligence and Applications*, pages 273–278. IOS Press, 2022.
- [ZPA+21] Xing Zhao, Manos Papagelis, Aijun An, Bao Xin Chen, Junfeng Liu, and Yonggang Hu. Zipline: an optimized algorithm for the elastic bulk synchronous parallel model. *Mach. Learn.*, 110(10):2867–2903, 2021.
- [ZPSP05] Xinwen Zhang, Francesco Parisi-Presicce, Ravi S. Sandhu, and Jaehong Park. Formal model and policy specification of usage control. *ACM Trans. Inf. Syst. Secur.*, 8(4):351–387, 2005.

## Titles in the IPA Dissertation Series since 2022

- A. Fedotov.** *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01
- M.O. Mahmoud.** *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02
- M. Safari.** *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03
- M. Verano Merino.** *Engineering Language-Parametric End-User Programming Environments for DSLs.* Faculty of Mathematics and Computer Science, TU/e. 2022-04
- G.F.C. Dupont.** *Network Security Monitoring in Environments where Digital and Physical Safety are Critical.* Faculty of Mathematics and Computer Science, TU/e. 2022-05
- T.M. Soethout.** *Banking on Domain Knowledge for Faster Transactions.* Faculty of Mathematics and Computer Science, TU/e. 2022-06
- P. Vukmirović.** *Implementation of Higher-Order Superposition.* Faculty of Sciences, Department of Computer Science, VU. 2022-07
- J. Wagemaker.** *Extensions of (Concurrent) Kleene Algebra.* Faculty of Science, Mathematics and Computer Science, RU. 2022-08
- R. Janssen.** *Refinement and Partiality for Model-Based Testing.* Faculty of Science, Mathematics and Computer Science, RU. 2022-09
- M. Laveaux.** *Accelerated Verification of Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2022-10
- S. Kochanthara.** *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01
- L.M. Ochoa Venegas.** *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02
- N. Yang.** *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03
- J. Cao.** *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04
- K. Dokter.** *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05
- J. Smits.** *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

- A. Arslanagić.** *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07
- M.S. Bouwman.** *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08
- S.A.M. Lathouwers.** *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09
- J.H. Stoel.** *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10
- D.M. Groenewegen.** *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11
- D.R. do Vale.** *On Semantical Methods for Higher-Order Complexity Analysis.* Faculty of Science, Mathematics and Computer Science, RU. 2024-01
- M.J.G. Olsthoorn.** *More Effective Test Case Generation with Multiple Tribes of AI.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02
- B. van den Heuvel.** *Correctly Communicating Software: Distributed, Asynchronous, and Beyond.* Faculty of Science and Engineering, RUG. 2024-03
- H.A. Hiep.** *New Foundations for Separation Logic.* Faculty of Mathematics and Natural Sciences, UL. 2024-04
- C.E. Brandt.** *Test Amplification For and With Developers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05
- J.I. Hejderup.** *Fine-Grained Analysis of Software Supply Chains.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06
- J. Jacobs.** *Guarantees by construction.* Faculty of Science, Mathematics and Computer Science, RU. 2024-07
- O. Bunte.** *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software.* Faculty of Mathematics and Computer Science, TU/e. 2024-08
- R.J.A. Erkens.** *Automaton-based Techniques for Optimized Term Rewriting.* Faculty of Mathematics and Computer Science, TU/e. 2024-09
- J.J.M. Martens.** *The Complexity of Bisimilarity by Partition Refinement.* Faculty of Mathematics and Computer Science, TU/e. 2024-10
- L.J. Edixhoven.** *Expressive Specification and Verification of Choreographies.* Faculty of Science, OU. 2024-11
- J.W.N. Paulus.** *On the Expressivity of Typed Concurrent Calculi.* Faculty of Science and Engineering, RUG. 2024-12
- J. Denkers.** *Domain-Specific Languages for Digital Printing Systems.* Faculty of Electrical Engineering,

Mathematics, and Computer Science, TUD. 2024-13

**L.H. Applis.** *Tool-Driven Quality Assurance for Functional Programming and Machine Learning.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

**P. Karkhanis.** *Driving the Future: Facilitating C-ITS Service Deployment for Connected and Smart Roadways.* Faculty of Mathematics and Computer Science, TU/e. 2024-15

**N.W. Cassee.** *Sentiment in Software Engineering.* Faculty of Mathematics and Computer Science, TU/e. 2024-16

**H. van Antwerpen.** *Declarative Name Binding for Type System Specifications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-01

**I.N. Mulder.** *Proof Automation for Fine-Grained Concurrent Separation Logic.* Faculty of Science, Mathematics and Computer Science, RU. 2025-02

**T.S. Badings.** *Robust Verification of Stochastic Systems: Guarantees in the Presence of Uncertainty.* Faculty of Science, Mathematics and Computer Science, RU. 2025-03

**A.M. Mir.** *Machine Learning-assisted Software Analysis.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-04

**L.T. Vinkhuijzen.** *Data Structures for Quantum Circuit Verification and How To Compare Them.* Faculty of Mathematics and Natural Sciences, UL. 2025-05

**D. van der Wal.** *What is the Point? Single-Input-Change Testing a EULYNX Controller.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-06

**A. Rosset.** *Uniform Monad Presentations and Graph Quasitoposes.* Faculty of Sciences, Department of Computer Science, VU. 2025-07

**L. Guo.** *Higher-Order Termination with Logical Constraints.* Faculty of Science, Mathematics and Computer Science, RU. 2025-08

**D.A. Manrique Negrin.** *A Model Orchestra in Digital Twins: A Model-Driven Approach to Integration and Orchestration.* Faculty of Mathematics and Computer Science, TU/e. 2025-09

**C.A. Esterhuysen.** *Specification-Centric Multi-Agent Systems.* Faculty of Science, UvA. 2025-10