Run, agent, run! Architecture and benchmarking of actor-based agents

Mohajeri Parizi, M.; Sileno, G.; van Engers, T.; Klous, S.

DOI
10.1145/3427760.3428339

Publication date
2020

Document Version
Final published version

Published in
AGERE '20

Citation for published version (APA):
Mohajeri Parizi, M., Sileno, G., van Engers, T., & Klous, S. (2020). Run, agent, run! Architecture and benchmarking of actor-based agents. In E. Castegren, J. De Koster, & T. C. Schmidt (Eds.), *AGERE '20: proceedings of the 10th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control : November 17, 2020, Virtual, USA* (pp. 11-20). The Association for Computing Machinery. https://doi.org/10.1145/3427760.3428339

Run, Agent, Run! Architecture and Benchmarking of Actor-Based Agents

Mostafa Mohajeri Parizi
m.mohajeriparizi@uva.nl
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

Giovanni Sileno
g.sileno@uva.nl
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

Tom van Engers
vanengers@uva.nl
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

Sander Klous
s.klous@uva.nl
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

Abstract

The paper introduces an Agent-Oriented Programming (AOP) framework based on the Belief-Desire-Intention (BDI) model of agency. The novelty of this framework is in relying on the Actor model, instantiating each intentional agent as an autonomous micro-system run by actors. The working hypothesis behind this choice is that defining the agents via actors results in a more fine-grained modular architecture and that the execution of agent-oriented programs is enhanced (in scalability as well as in performance) by relying on robust implementations of Actor models such as *Akka*. The framework is benchmarked and analyzed quantitatively and qualitatively against three other AOP frameworks: Jason, ASTRA and Sarl.

CCS Concepts: • **Computing methodologies** → *Modeling and simulation*; • **General and reference** → *Evaluation*; *Performance.*

Keywords: Agent-Oriented Programming, Reactive Programming, Intentional Agents, BDI, Actor Model, Benchmark

ACM Reference Format:
Mostafa Mohajeri Parizi, Giovanni Sileno, Tom van Engers, and Sander Klous. 2020. Run, Agent, Run! Architecture and Benchmarking of Actor-Based Agents. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3427760.3428339

AGERE '20, November 17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8185-7/20/11…$15.00
https://doi.org/10.1145/3427760.3428339

1 Introduction

Agent-based models have an intuitive mapping to behavioural descriptions, and for this reason are extensively used for modeling and simulations of social systems. However, *agent-based programming* is not only relevant for simulation. Data-sharing infrastructures as digital marketplaces exhibit the double status of computational and social systems; regulating these infrastructures requires reproducing to a certain extent constructs similar to those observed in human reasoning (e.g. For which purpose is the agent asking access to the resource? On which basis is the infrastructure granting access?). For *traceability* and *explainability* reasons, decisions concerning actions need to be processed by the infrastructure as much as relevant operational aspects. Agent-based programming, by looking at computational agents as intentional agents, provides this level of abstraction available *by design*. However, this raises concerns on how we can efficiently map logic-oriented agent-based programs into an operational setting, a problem motivating the present research.

This paper introduces AgentScriptCC, a logic-based AOP framework in which agents are modular micro-systems run by actors. To evaluate the feasibility of this approach for future developments, a first implementation of AgentScriptCC based on Akka runnig on JVM is compared with three other relevant AOP frameworks (Jason [4], ASTRA [13] and Sarl [13]) by means of 3 benchmarks (token ring, chameneos redux and service point), known to capture relevant patterns in concurrent applications. This performance evaluation shows that despite its relative youth and the new implementation approach, AgentScriptCC is competitive against existing frameworks, making it worthy of further investigation.

The paper proceeds as follows: Section 2 provides some background on relevant concepts and related works. Section 3 presents the AgentScriptCC framework. Section 4 reports on the empirical experiments comparing AgentScriptCC with other frameworks. Section 5 compares the frameworks qualitatively. A note of future developments ends the paper.

2 Background

2.1 Agent Oriented Programming

Agent-Oriented Programming (AOP) is a programming paradigm that uses mental attitudes to model autonomous computational agents. Introduced in 1993 by Shoham [27], it has attracted increasing attention ever since and is believed to provide an effective abstraction to approach complex software systems (e.g. [26]). In the beginning it was presented as a specific version of Object Oriented Programming (OOP): whereas object classes contain arbitrary components, agent types share the same types of mental states and of structural relationships/mechanisms involving those states.

2.2 Belief-Desire-Intention (BDI) Model

Having its roots in a theory of mind [5], and so referring to categories that are used typically to address human behaviour to describe agents, the *belief-desire-intention* (BDI) model [25] has been extensively investigated as basis to represent computational agents that exhibit rational behaviour [16]. *Beliefs* are the factual (and possibly inferential) information the agent has about itself or its environment. *Desires*, in their simplest form, are objectives the agent wants to accomplish. *Intentions* are the courses of action the agent has committed to. In practice, BDI agents include concepts of *Goals* and *Plan*. Goals are instantiated desires and plans are abstract specifications relating a goal to the means of achieving that goal (intentions become commitment towards plans). Multiple programming languages and frameworks have been introduced based on the BDI model, as AgentSpeak(L)/Jason [4, 24], 3APL/2APL [10], GOAL [19] and IMPACT [14].

2.3 Actor Model

The Actor model, introduced in [18], is a mathematical theory that treats *actors* as the primitives of computation [17]. Actors are essentially reactive concurrent entities. When an actor receives a message it can send messages to other actors; *spawn* new actors; modify its reactive behavior for the next message it receives. Originally proposed as a tool for the theoretical understanding of concurrency, the Actor model serves now as the basis of several production-level solutions for distributed and asynchronous systems, and for reactive programming. These solutions include: Akka [15], a library developed for the JVM environment, enriched by a strong community with multiple complementary tools for distributed environments and stream processing; the C++ Actor Framework (CAF) [7], a library for creating concurrent programs in C++; Pony [8, 9], an actor language for building robust parallel systems by providing data-race free isolation for actors. A comprehensive overview and benchmark over these works can be found in [3].

2.4 Related Work

Multiple AOP and BDI frameworks have been introduced proposing diverse approaches towards language, execution model, reasoning process, etc. Jason [4] is plausibly the most known (e.g. it is the most used choice in the Multi-Agent Programming Contest [11]), and has been constantly developed in the last 15 years. It is implemented in Java and is essentially an *interpreter* for a logic-based DSL, namely an extended version of AgentSpeak(L) [24]. Two recent frameworks inspired by Jason are Pyson [1] and LightJason [2]. Pyson is an interpreter implemented in Python and uses `MapReduce` technology as execution infrastructure in order to achieve better scalability specifically w.r.t. the number of agents. LightJason is a BDI framework based upon a variation of AgentSpeak(L) and whose interpreter aims to improve the scalability of Jason by implementing a concurrent platform following best practices in software engineering.

ASTRA [13] is yet another framework inspired by AgentSpeak(L)/Jason and is also implemented in Java, but, unlike Jason, it is not an interpreter. ASTRA relies on a *compilation* approach: through a build pipeline the DSL is first translated to pure Java code and then the Java code is compiled to bytecode for execution. In contrast, the Sarl [26] framework has not been introduced as a BDI platform and then it does not use the same abstractions. Nonetheless it is an AOP framework written in Java that also uses compilation, and for these reasons it is relevant for the current study.

Although several AOP/BDI frameworks have been introduced in the recent years (all hinting to problems of scalability), there is a small amount of empirical data available about how they perform in comparison to each other. The most notable exception is [6], in which multiple actor and agent frameworks (2APL [10], GOAL [19], Jason and Akka) are benchmarked. Their results showed that Jason outperformed other BDI frameworks by far and scaled almost on par with Akka. However, at that time (2013), none of these newer frameworks had been introduced yet, and Akka had not the support it has today. Strangely enough, none of these new AOP frameworks has the Actor model at their foundation. The present paper aims to investigates part of this gap.

3 AgentScriptCC

The AgentScriptCC framework consists of: (a) a logic-based Agent-Oriented Programming DSL; (b) an abstract execution architecture; (c) a translation method that generates executable models from models specified by the DSL; (d) tools that support the execution of models. We provide here a brief overview on these components.

3.1 AgentScriptCC DSL

The AgentScriptCC DSL has a very close syntax to AgentSpeak(L) language and includes some of the extensions provided by Jason. The main components of the DSL are (1) initial beliefs, (2) inferential rules, (3) initial goals, and (4) plan rules. The initial beliefs and goals express the mental state of the agent at the start of the execution. Initial beliefs are

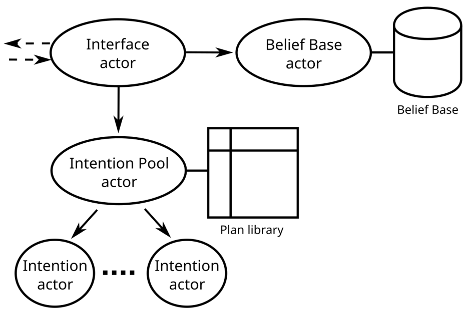

Figure 1. AgentScriptCC execution architecture

a set of Prolog-like facts, and the initial goals designate the first intentions to which the agent commits. Inferential rules are potentially non-grounded declarative rules (Prolog-like), used to infer beliefs from beliefs. Plan rules are potentially non-grounded reactive rules in the form $e : c \Rightarrow f$ that map different internal events (e.g, *goal adoption*, *belief-update*) or external events (e.g, *message reception*, *perception*) to a sequence of executable steps called the *plan body* which the agent has to perform in response to the event. When a plan body f is matched with an event e, it is said that f is *relevant* for e. Each plan also has a context condition c which is a Prolog-like expression. When a plan f is relevant for e and also c holds, it is said that the f is *applicable* for c. The steps of a plan body can include belief query, belief update, sub-goal adoption, primitive actions, variable assignment, and control flow structures (loops and conditionals).

3.2 AgentScriptCC Execution Architecture

In contrast to AgentSpeak(L)/Jason, the execution architecture of AgentScriptCC agents is based on the Actor model. Each agent consists of multiple actors with different roles: (i) an **Interface** actor, (ii) a **Belief Base** actor, (iii), an **Intention Pool** actor and (iv) $N \geq 1$ **Intention** actors. Each agent has also non-actor components: (1) a plan library, and (2) one or more belief bases.

The plan library of the agent consists of a set of plan rule objects in the form `{e,c,f}`, where `e` is an object that can be matched and unified with event messages to determine if a plan rule is relevant for that event, `c` is an expression object that can be sent to the Belief Base actor to determine if the plan is applicable and `f` is a function representing the body of the plan.

The belief base(s) of the agent can be in practice any type of storage technology. To interface an arbitrary belief base into the agent architecture a translation function needs to be implemented for mapping the query messages into the queries of that belief base and vice versa, translating the responses into result messages.[1]

3.2.1 Interface Actor.

The Interface actor acts as the main entity of the agent. It initializes the Belief Base and Intention Pool actors and then sends the initial beliefs and inferential rules to Belief Base actor as *assert* messages and initial goals to Intention pool actor as *achieve* messages. This actor is the only component of the agent that is accessible from the environment and the other agents: all incoming messages and events must go though this actor and any message sent from this agent will indicate the Interface actor as the sender of message. When the Interface actor receives a new message m, based on the type of the message it will either process it itself if m is a control messages, (e.g, *halt*), forward it to Belief Base actor if m is an assert message (e.g, *perception*) or forward it to Intention Pool actor if m is an achieve message (e.g, *request*).

3.2.2 Belief Base Actor.

The Belief Base actor maintains the connection between other components of the agent and any data storage/reasoning engine that is used as the belief base. This actor accepts query messages (*retract*, *assert* and *unify*) and responds with result of the query. The technology of the data storage(s) is abstracted behind this actor and it can be changed by the programmer without affecting the rest of the framework.Apart from processing queries, the Belief Base actor also feeds back *belief-update* events to the Interface actor. The semantics of when these events should be created are externalized to the core of architecture and can be programmable by the designer.

3.2.3 Intention Pool actor.

The Intention Pool actor receives events from the Interface actor and processes them. To process a received event `v`, the set of relevant plan rules `{e,c,f}` are selected from the plan library by matching and unifying `v` against `e`. Then these relevant plans are fetched from the plan library and sent to an idle Intention actor. The Intention pool actor can spawn N Intention actors, where the configurable number N dictates the number of concurrent intentions each agent can have at each instant. This actor uses a prioritized mailbox that sorts the messages based on the externalized programmable priority function S_E and a new event is processed only if there are idle Intention actors to forward it to. This mechanism makes sure that as long as there are no resources available, new events stay in the mailbox to be re-prioritized by S_E and when an idle Intention actor becomes available the event with the highest priority is processed [2].

3.2.4 Intention Actor.

An Intention actor is a reusable unit of execution for the agent. It receives an event `v` alongside a set of plan rule objects `{e,c,f}` from the Intention Pool actor for execution. The execution consists of three

[1]For the benchmarks presented in this work we used a lightweight open-source Prolog reasoning engine implemented in Scala called *Styla*, available at https://github.com/fedesilva/styla. The library was minimally modified and is available at https://github.com/mostafamohajeri/styla.

[2]In the current implementation, the Intention pool actor exploits the *Router* feature of Akka.

phases: (i) the applicability of each plan rule is checked by sending a query message containing `c` to the Belief Base actor; (ii) from the set of applicable plans, one is selected by the externalized programmable function S_P for execution; (iii) the function `f` of the selected plan is executed by the Intention actor. After the execution of `v` is completed either by success or failure status, a message is sent to the actor which originally requested `v` containing the completion status and also a message is sent to Intention Pool actor signaling that this actor is now idle.

3.3 Translation Method

The translation method is designed to compile the models specified with the AgentScriptCC DSL described in 3.1 into agents with the architecture described in 3.2.

For each entity of the DSL, a mapping is defined to generate the code in the executable underlying language that can instantiate the objects with the desired semantics at runtime. The translated entities are then fitted in the abstract architecture to form an executable agent program.

3.3.1 Terms and Expressions. The AgentScriptCC DSL uses Prolog-style terms and expressions. In the translation of an script written in the DSL, each term and expression (including inferential rules) maps to a `Term` or `Expression` object which encapsulates the parsed data (potentially containing nested `Terms` and `Expressions`).

Access to the Lower-Level Language. As consequence of an approach based on compilation, the DSL provides direct access to any object or function available in the agent's name space[3]. These lower-level access statements, indicated by the token #, are translated literally to the same statement in the underlying language. This capability provides fast and seamless reuse of libraries already established for the underlying language.

3.3.2 Initial Beliefs/Goals and Inferential Rules. At syntactic level, initial beliefs and inferential rules are logic-style expressions, and as such they translate to an `Expression` object counterpart. Initial goals are a combination of a prefix (`!`, designating the adoption of a new goal) and a term and they translate to a `Goal` object encapsulating the prefix and a `Term` object.

3.3.3 Plan Rules. A plan rule $< e, c, f >$, should be translated into the object `{e,c,f}` which will be part of the plan library. The triggering event of the plan rule e consists of a trigger (one of `+!`,`-!`,`+?`,`+`,`-`) and a term t. The triggers convey the relevance of the plan to different event types while t can be seen as the payload of that event; `+!` relates to adoption of a new goal, `-!` relates to failure of a goal, `+?` relates to testing if a term holds true, `+` and `-` respectively relate to assertion and retraction of a belief. The triggering event e then translates to an `Event` object which encapsulates the trigger and the translated `Term` object of t. The context condition c is an expression and translates to an `Expression` object. The plan body f of a plan rule consist of zero or more steps. It is translated into a function `f`, which contains the steps of f as imperative lines of code implemented in it. Each type of step is translated differently as is described below.

Primitive Actions. A primitive action of the form `#f(...)` is translated into a lower-level call to a function `f` defined in the underlying language with its respective parameters.

Variable Assignments. Variable assignments in form of `V = exp` are used to (re-)assign the result value of an expression `exp` to a variable `V`. AgentScriptCC uses an internal map-like approach to store variables that also manages variable scopes, meaning that each code block (e.g, plan body, condition block) holds a map of all variables declared in that scope which also inherits the variables in its parent scope. A variable assignment is translated to an append operation for the variable map by using the V as the key and `exp` as the value.

Belief Queries. Belief query steps are composed of a prefix `+`,`-` and a term t. The prefixes respectively mean assertion and retraction. As the belief base of the agent is abstracted by the Belief Base actor, a belief query step is a blocking message to the Belief Base actor containing the prefix and the `Term` object of t.

Sub-Goal Adoption. Task decomposition is crucial component of BDI-like agents and in essence is the ability to adopt sub-goals depending on the context of a plan. At the syntactic level, a (sub-)goal adoption is a prefix (e.g, `!`,`?`) plus a term t. The prefixes respectively mean achievement and test goals. In the translation method a sub-goal adoption step is translated as two phases, (i) a plan selection by using S_P is done to select and fetch a plan rule object `{e,c,f}` from the plan library, (ii) the function `f(...)` is called with any parameters that t may have as the arguments of `f`.

Control Flow Structures. The compilation method of AgentScriptCC supports a straightforward mapping of simple control flow structures such as loops and conditionals to their executable counterparts. The translation of these control structures to the underlying language is performed one-to-one; for example an `if/else` in the DSL is simply translated to an `if/else` in the underlying language.

3.4 Tools for Execution

The architecture of AgentScriptCC agents is based on actors and for their execution these actors require an *actor system* that *spawn* and *start* them. Additionally, a message *transportation layer* needs to be specified to enable communication between agents. The framework remains agnostic

[3] In the Scala implementation, any object or function which is accessible via the Java class path.

with respect the transportation layer as long as there is an interface to convert messages from and to AgentScriptCC's message protocol.

Our current implementation of AgentScriptCC is written in Scala and is based on the Akka framework. In addition to a compiler[4], it includes a minimal infrastructure that is able to spawn and start the compiled agents[5]. The transportation layer makes simply use of Akka's typed messages, but other solutions can be easily integrated.

4 Benchmarks

The following section proposes quantitative comparisons between the AgentScriptCC framework and three other frameworks: Jason (`v2.5`), ASTRA (`v1.0.0`) and Sarl (`v0.11.0`). Jason [4] was chosen because, like AgentScriptCC, it uses a language based on AgentSpeak(L), is implemented in Java and as reported by [6] potentially outperforms other BDI frameworks. ASTRA and Sarl are both also implemented in Java, but, more importantly, like AgentScriptCC, rely on a compilation approach.

Performance comparison is effectuated by means of two fairly standard benchmarks (token ring, chameneos redux), close to what has been presented in [6]. The main difference w.r.t. [6] is the metrics, as we separate the interpretation/setup time from the execution time, to present better insights on how these frameworks operate. An additional benchmark (service point) was also performed to assess the ability of the frameworks to allow concurrent decomposition of tasks inside the agents. The benchmarks were performed on a Debian GNU/Linux 10 machine with an 8 core Intel(R) Xeon(R) CPU E5-1620 v4 @ 3.50GHz CPU and 64GB of RAM using Java version 11 with GraalVM 20 JRE. Each benchmark was performed 10 times and the JVM was stopped between each run to avoid the impact of one experiment on the next.

In the first two benchmark scenarios, three metrics are recorded: (1) total interpretation/setup time, including agent creation time, (2) internal execution time measured from the instant that the first agent starts until the completion of the test, and (3) CPU core load. Execution and data gathering is controlled by a Python script that runs the benchmarks in the desired dimensions and records the metrics[6].

4.1 Token Ring

The token ring benchmark is a simple program targeting multiple aspects of parallel frameworks: handling different number of agents, message passing and level of concurrency each agent can achieve. The testing scenario consists of W worker agents, T tokens are distributed among the workers, and each token has to be passed N times in a ring. When all T tokens have been passed N times, the program ends. To run this benchmark a program should:

- create W number of workers;
- each worker should be connected to its neighbor forming a complete ring;
- initially each token $1 \leq i \leq T$ is assigned to a worker $1 \leq j \leq W$ with the equation $j = i * (W/T)$
- each worker sends the token to its neighbor

The program finishes when all T tokens have been passed N times.

The experiment was performed by varying W, T and N independently within the values $\{4, 16, 256, 1k, 4k\}$, resulting in 125 different configurations for each framework. We also put a 1 minute limit for each execution and anything beyond that is considered a *timeout*.

4.1.1 Implementation Notes.

In all implementations a *broker* agent is present that starts the benchmark by distributing the tokens and gathers the completed tokens to stop the execution. There is a difference in the Sarl implementation. As Sarl does not provide a central agent resolver to address agents by name, an extra step is implemented in the broker to iterate over all worker agents and link them together in a ring.

4.1.2 Results.

A summary of the results for this benchmark is presented in Figures 2 and 3. In Figure 2, the number of agents W is the variable while N and T are kept constant with two settings ($N = 256, T = 256$) and ($N = 4k, T = 4k$). Only Jason and AgentScriptCC were able to execute ($N = 4k, T = 4k$). Sarl was able to only execute the benchmark up to $W = 256$ agents and timed out with a warning[7]. ASTRA seemed stable enough to finish the ($N = 4k, T = 4k$) test but not within 1 minute. ASTRA executes very poorly for ($N = 256, T = 256$) test, especially with lower number of worker agents, plausibly because with less worker agents each agent has more concurrent threads of work to execute. AgentScriptCC and Jason both perform almost without much effect w.r.t. number of agents, suggesting that both frameworks can handle concurrency inside agents to a good extent, although in all cases Jason performs marginally better.

In Figure 3 another view on the results is presented. This time the variable is the number of tokens T, whereas W, N are kept constant in two settings: ($W = 256, N = 256$) and ($W = 4k, N = 4k$). Like in the previous results Sarl could only finish the ($W = 256, N = 256$) test. ASTRA was able to execute the ($W = 4k, N = 4k$) test but only up to $T = 1k$ and timed out after that. In the ($W = 256, N = 256$) Jason and AgentScriptCC performed much better and scaled almost linearly with the number of tokens which shows that both

[4]Source code: https://github.com/mostafamohajeri/scriptcc-translator.

[5]Source code: https://github.com/mostafamohajeri/agentscript.

[6]Source code: https://github.com/uva-cci/aop-benchmarks-agere2020.

[7]Potentially dangerous stack overflow in `java.util.concurrent.locks.ReentrantReadWriteLock`. We suspect this occurs because at the start all workers need to send a message to the broker to get their neighbors and the broker can not handle this amount (≥ 1024) of concurrent messages.

frameworks can handle the increased concurrency and the higher number of messages to be passed in an efficient manner. On the other hand Sarl and ASTRA performed poorly under the increasing amount of tokens. In the ($W = 4k, N = 4k$) test Jason performs marginally better than AgentScriptCC.

CPU Load. Figure 4 and Figure 5 present the average core load during the token ring test respectively in the $W, T = 256$ and $N = 4096$ and in the $W, T, N = 4k$ settings. In the lower settings (Figure 4) Jason and ASTRA have much less CPU demand than AgentScriptCC and Sarl. On the other hand, in the higher setting (Figure 5) the CPU load between Jason and AgentScriptCC is closer (respectively 85.7% and 88.6%, vs 57.7% and 77.7% in the lower setting). This can be an indication that AgentScriptCC has a higher footprint on the CPU load, especially for initialization time.

To understand how much each framework can distribute the load between CPU cores we have to look at the standard deviation of CPU load data. A higher deviation indicates that the framework is not balancing the load between cores. ASTRA shows to have very poor load balancing with the deviation almost as high as the average which can mean that some of the cores are not even used in execution. Sarl has a high balancing of cores even in lower setting. In the higher settings both Jason and AgentScriptCC seem to distribute the load between CPU cores sufficiently.

Initialization Time. To assess the initialization time, total execution time is subtracted by the internal execution time in the lowest setting with $N = 4k$ and $T = 4k$ and the results are presented for an increasing number of agents in Figure 6. ASTRA proves to have the fastest initialization, at least up to $4k$ agents, followed by Jason and closely by AgentScriptCC. Sarl seems to have the slowest initialization time and scales very badly with the number of agents.

4.2 Chameneos Redux

The second benchmark is adopted from [20] and is a test intended to capture the effects of one limiting point to the execution framework. The scenario consists of C chameneo creatures living in the jungle; they can go to a common place to meet other creatures and *mutate* with them. Each creature has a color assigned to it from a color pool and after mutation its colour changes based on the color of the other creature it met. These meetings should happen for a total number of N times. To run this benchmark a program should:

- create C differently colored (blue, red, yellow), differently named, concurrent chameneo creatures
- write all the possible complementary color combinations;
- write the initial color of each creature;
- each creature will repeatedly go to the meeting place and meet, or wait to meet, another chameneo;

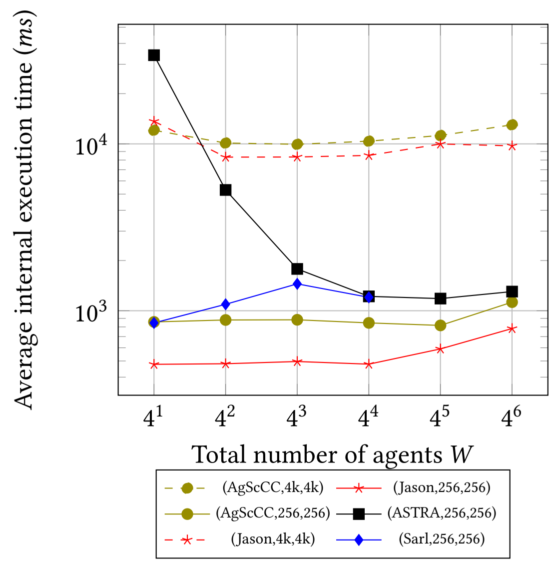

Figure 2. Token ring results for each (framework, T, N)

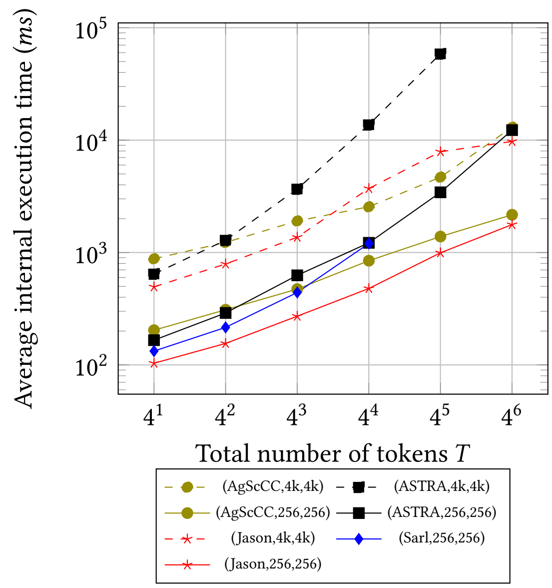

Figure 3. Token ring results for each (framework,W, N)

- both creatures will change color to complement the color of the chameneo that they met;
- after N meetings have taken place, for each creature write the number of creatures met and the number of times the creature met a creature with the same name (should be zero).
- the program finishes when N meetings have happened.

The experiment was performed with the set of variables $C = \{64, 256, 1k, 4k\}$ and $N = \{1k, 4k, 16k, 64k\}$. This provide us with 20 different configurations for each framework. All

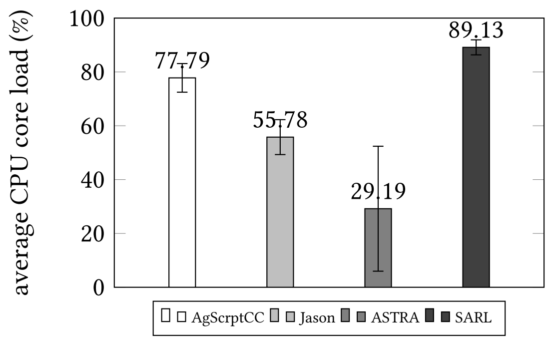

Figure 4. CPU load (average and standard deviation on 8 cores) in token ring with $N = 4k$, $T = 256$ and $W = 256$

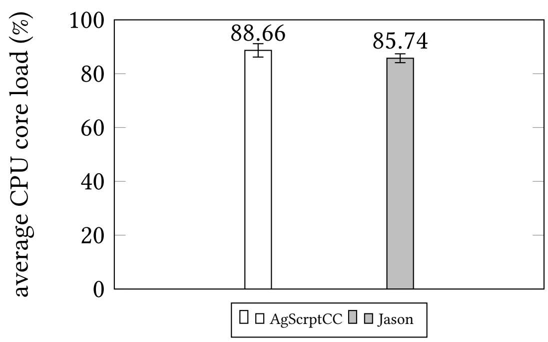

Figure 5. CPU load (average and standard deviation on 8 cores) in token ring with $N = 4k$, $T = 4k$ and $W = 4k$

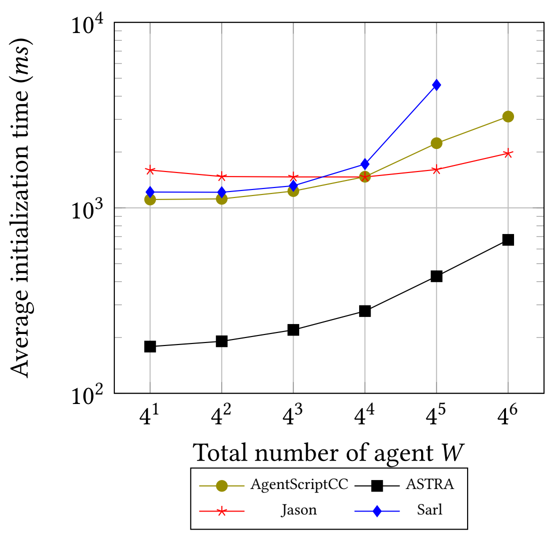

Figure 6. Initialization time in token ring with $T = 4$, $N = 4$

tests were given a 1 minute time limit and it is considered a timeout after that.

4.2.1 Implementation Notes. In all implementations a *broker* agent is present that acts as the meeting point for chameneos. This agent is the main point of this benchmark

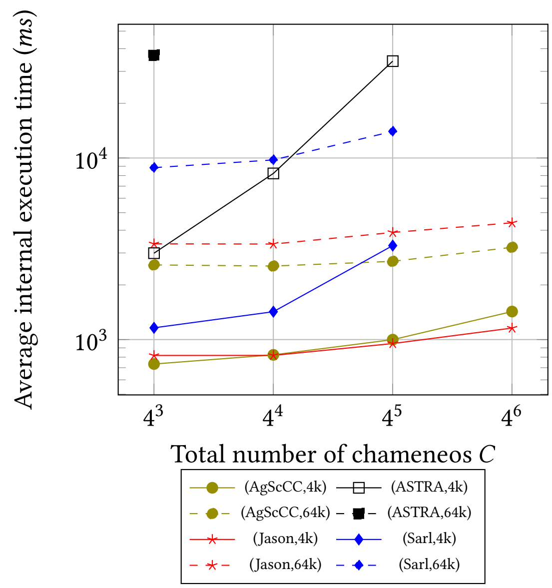

Figure 7. Chameneos redux results for each (framework, N)

as it will be constantly under high number of requests from the chameneos agents.

4.2.2 Results. The first view on the results is presented in Figure 7. In this setting the number of meetings N is kept constant at two values $4k$ and $64k$ whilst the number of chameneos is the variable. The results show that Jason and AgentScriptCC scale well with the number of agents while AgentScriptCC performs marginally better in the $N = 64k$ test. Sarl and ASTRA suffer from the higher number of agents to the point that Sarl could finish both tests only up to $C = 1k$ agents while ASTRA finishing $N = 64k$ test only in the $C = 64$ agents setting.

Figure 8 presents another view on the results. This time the number of chameneos C is kept constant at 256 and $4k$, whilst the number of meetings N is the variable. Sarl could only finish the $C = 256$ test while ASTRA could only finish it up to $N = 16k$ and timing out after that. ASTRA was also only able to finish the $C = 4k$ test with $C = 64$ number chameneos. AgentScriptCC and Jason both completed the tests with linear scaling, with AgentScriptCC outperforming Jason slightly in the $C = 4k$ test. This shows that both Jason and AgentScriptCC can handle higher levels of concurrency in the broker agent w.r.t. the increasing number of concurrent requests.

4.3 Service Point

This last benchmark is not about performance. Rather, it is designed to illustrate the differences between the execution in a *step-based* framework like Jason in contrast to a compilation-based framework like AgentScriptCC, focusing

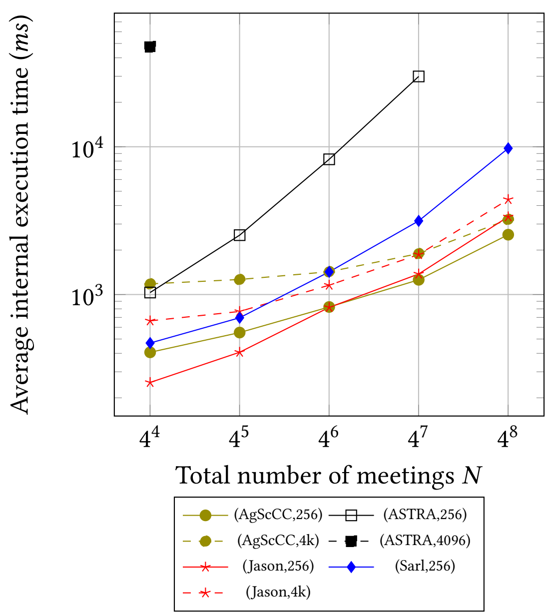

Figure 8. Chameneos redux results for each (framework,C)

on how they handle actions (namely time-consuming primitive actions) specified outside their DSL. The scenario of this benchmark consists of one service point and N number of consumers. Each consumer sends R requests to the service point and waits for the response. The service point needs a random amount of time t ($0 \leq t \leq 5000$ ms) to process each request. A simple `Thread.sleep(t)` is used to mimic thread time consumption. To run this benchmark a program should

- create 1 service point and N service consumers.
- each consumer will send R number of requests to the service point
- the program finishes when all of the $R * N$ requests have been responded

The experiment was done only on Jason and AgentScriptCC with variables $N = \{1, 4, 16\}$ and $R = \{1, 4, 16\}$. With respect to total number of request $R * N$, this gives us with 5 unique configurations. To account for the non-determinism added by the randomization each configuration is executed for 100 times.

4.3.1 Results. The results of this experiment are presented in Figure 9. Jason performs much worse in this scenario, as it is not being able to finish the 256 requests within a 200 seconds timeout. This is even more strange as in our setting Jason is set to use 8 threads and AgentScriptCC to 6 and by looking at the results we can see that AgentScriptCC is always using the thread times completely but Jason is not. The reason for this is that Jason uses a *sequential* reasoning cycle inside each agent; at every reasoning cycle, a Jason agent takes the next step from each of its intentions and executes them. The reasoning cycle ends when all intentions

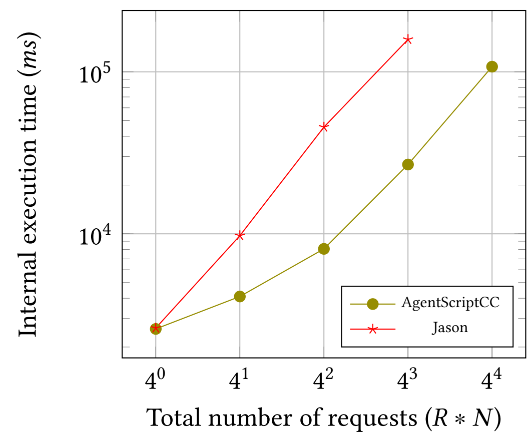

Figure 9. Service point scenario results

execute one step. This means that if in the reasoning cycle of an agent one of these steps is a time-consuming primitive action, the whole cycle will be blocked[8]. On the contrary a compiled agent does not have any notion of steps at run-time and the parallelism between intentions of the agent is also handled by the underlying concurrency model, in this case the Actor model. This matter is further discussed in 5.3.

5 Discussion

This paper presents and evaluates a framework for an AOP language based on AgentSpeack(L) relying on compilation. Compilation in this context is not novel as it has been used previously by other AOP frameworks like SARL [26] and ASTRA [13]. The novelty of this work lies in two aspects. First, unlike SARL and ASTRA, that use a DSL very close to their underlying language (Java), AgentScriptCC uses a logic-based DSL close to AgentSpeak(L). As our pipeline starts from an *antlr* grammar, in principle the current DSL can be replaced by any other AOP language that can be mapped to the AgentScriptCC abstract execution architecture. Second, our approach maps the DSL into an architecture that exploits the Actor model. This means that not only the final executable model is more robust, because it takes advantage of the established concurrency model and the maturity of the libraries implementing the Actor model (e.g, Akka), but also that the translation itself is an open process, so its product becomes in principle more understandable for the programmer.

5.1 Performance

The execution model of AgentScriptCC is closer to Sarl and ASTRA than to Jason (see 5.3), but, as shown by the benchmarks, it is substantially outperforming both Sarl and ASTRA. At the same time, AgentScriptCC performance was

[8] Jason provides extra built-in directives like `.wait` to mimic unblocking suspension of intentions but that is beyond the context of this benchmark.

below what we expected before running these experiments. Investigating possible causes by profiling the execution of benchmarks, we found out that a considerable amount of execution time is spent on the blocking due to *synchronized query calls* to the belief base. These calls had to be synchronized because Prolog engines like *Styla* and *tuProlog* [12] (another candidate solution we tested for handling belief bases) are inherently made for single thread access. Even a simple *read* query to a Prolog engine still counts as *write* access because of the backtracking. We believe once this issue is addressed the performances of AgentScriptCC will greatly improve.

5.2 Language

Although all of the considered frameworks propose DSLs to program reactive agents, their bases are different. AgentScriptCC's DSL is based on AgentSpeak(L), which gives to the language a logic-oriented flavour; this is also the case for Jason, and both frameworks can take advantage of Prolog-style terms and expressions. ASTRA's DSL is also based on concepts defined in AgentSpeak(L) but with more syntactic resemblance to Java. Sarl's language does not try to be a logic-based language, therefore it does not contain components corresponding to terms or expressions; it is rather very close to Java.

5.3 Execution and Parallelism

As a common ground, all these frameworks are used to specify reactive agents, but they differ in how the agent's (re)actions are executed. The most particular solution comes with Jason which uses the concept of *steps*. The Jason interpreter treats each symbolic step/instruction in a plan of a reactive rule as a single unit of execution, and emulates an imperative program by executing them in a sequence in consecutive reasoning cycles. In contrast, in the other three frameworks, the steps of the reactive rules are already imperative programs ready to be executed. The approach taken by Jason has important consequences especially when agents execute multiple parallel threads of work (intentions) at the same time. This concept is examined more in detail in section 4 and in [13].

5.4 Access to the Lower-Level Language

One of the motivations behind developing AgentScriptCC has been to enable access to libraries defined in the underlying general-purpose programming language of choice in a easy and seamless way. In our view this impacts the usability of the framework in larger applications. Leading by an example, consider a programmer that needs to call the Java function `Thread.sleep(1)` in a reactive rule. In Jason one needs to create an extra class extending one of Jason's internal classes (`Agent`, `Action` or `Environment`) and define a method that wraps this low level function and then call the wrapping method from the agent program. In ASTRA it is almost the same as Jason and one needs to create a class extending the type `Module`, wrap this function inside a method, and annotate it appropriately to be able to call it from the agent program. On the opposite side, this is entirely different for Sarl and AgentScriptCC, as one can simply call this function directly from the agent program. In case of AgentScriptCC this can be done with `#Thread.sleep(1)`.

5.5 Communication

The communication in AgentScriptCC is entirely externalized, both for agent-to-agent and agent-to-environment communication. In the current implementation communication between agents uses Akka's internal message system but this can easily be replaced with any other type of communication mechanism, e.g. by using a message queue (*MQ*) to be able to execute the agents in a distributed setting. For the other frameworks, externalization is possible, but requires specific wrappers to the communication infrastructures (Jason with JADE).

6 Conclusion and Future developments

The slowly but steadily increasing interest in programming languages based on BDI or functionally similar architectures for virtual assistants, robotics, (serious) gaming, as well as for social simulations, hints that there is a general consensus that these solutions might be suitable to reproduce human-like reasoning, or rather human-intelligible computation.

Historically, the majority of contributions in this area were concerned mostly by the logical aspects of the problem rather than its computational aspects [16]. However, more recent contributions revealed the presence of issues w.r.t. computational performance and compatibility to modern environments and tools, motivating efforts to redevelop existing BDI frameworks according to best practices [1, 2]. Looking at intentional programming in the longer term, we need to acknowledge that operational settings differ from the typical low-scale simulation setting in which it is used today. Besides a difference in scale, components can also be fully distributed. Because of this, a future target feature of AgentScriptCC will be the capability to deploy and execute agents in distributed settings. This seems to be an achievable objective as there are already approaches available to run actors in distributed environments.

An initial, additional motivation of using an actor-oriented architecture for the intentional agents is that by having this extra level of abstraction the agent become more modular, enabling the augmentation of agents with complementary machinery like using AI modules [28], normative reasoning modules [22], planning (e.g. HTN, STRIPS) modules [21] and preference checking modules [23, 29]. Defining adequate interfaces to support the different types of add-ons for AgentScriptCC agents will be investigated in the future.

The present work acts as a starting block towards this path. The benchmarks reported here demonstrates that, despite the initial maturity level of the framework, AgentScriptCC is already competitive against existing frameworks, motivating further exploration.

Acknowledgments

This paper results from work done within the NWO-funded project *Data Logistics for Logistics Data* (DL4LD, www.dl4ld.net). The DL4LD project is funded by the Dutch Science Foundation in the Commit2Data program (grant no: 628.001.001).

References

[1] Tobias Ahlbrecht, Jürgen Dix, and Niklas Fiekas. 2017. Scalable Multi-agent Simulation Based on MapReduce. 364–371.

[2] Malte Aschermann, Philipp Kraus, and Jörg P. Müller. 2016. LightJason - A BDI Framework Inspired by Jason. In *EUMAS/AT*.

[3] Sebastian Blessing, Kiko Fernandez-Reyes, Albert Mingkun Yang, Sophia Drossopoulou, and Tobias Wrigstad. 2019. Run, Actor, Run: Towards Cross-Actor Language Benchmarking. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Athens, Greece) *(AGERE 2019)*. Association for Computing Machinery, New York, NY, USA, 41–50.

[4] Rafael H. Bordini, Jomi F. Hübner, and Renata Vieira. 2005. Jason and the Golden Fleece of Agent-Oriented Programming. In *Multi-Agent Programming: Languages, Platforms and Applications*. 3–37.

[5] Michael E. Bratman. 1987. *Intention, Plans, and Practical Reason*. Vol. 10. Harvard University Press.

[6] Rafael C. Cardoso, Maicon R. Zatelli, Jomi F. Hübner, and Rafael H. Bordini. 2013. Towards Benchmarking Actor- and Agent-Based Programming Languages. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Indianapolis, Indiana, USA) *(AGERE! 2013)*. Association for Computing Machinery, New York, NY, USA, 115–126.

[7] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2014. CAF - the C++ Actor Framework for Scalable and Resource-Efficient Applications. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control* (Portland, Oregon, USA) *(AGERE! '14)*. Association for Computing Machinery, New York, NY, USA, 15–28.

[8] Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully Concurrent Garbage Collection of Actors on Many-Core Machines. *SIGPLAN Not.* 48, 10 (Oct. 2013), 553–570.

[9] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Pittsburgh, PA, USA) *(AGERE! 2015)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[10] Mehdi Dastani. 2008. 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems* 16, 3 (2008), 214–248.

[11] Mehdi Dastani, Jürgen Dix, and Peter Novak. 2005. The First Contest on Multi-Agent Systems Based on Computational Logic. In *Proceedings of the 6th International Conference on Computational Logic in Multi-Agent Systems* (London, UK) *(CLIMA'05)*. Springer-Verlag, Berlin, Heidelberg, 373–384.

[12] Enrico Denti, Andrea Omicini, and Alessandro Ricci. 2001. Tu Prolog: A Light-Weight Prolog for Internet Applications and Infrastructures. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL '01)*. Springer-Verlag, Berlin, Heidelberg, 184–198.

[13] Akshat Dhaon and Rem W. Collier. 2014. Multiple Inheritance in AgentSpeak(L)-Style Programming Languages. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*. Association for Computing Machinery, New York, NY, USA.

[14] Jürgen Dix and Yingqian Zhang. 2005. *Impact: A Multi-Agent Framework with Declarative Semantics*. Vol. 15. 69–94.

[15] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410, 2 (2009), 202 – 220. Distributed Computing Techniques.

[16] Andreas Herzig, Emiliano Lorini, Laurent Perrussel, and Zhanhao Xiao. 2017. BDI Logics for BDI Architectures: Old Problems, New Perspectives. *KI - Künstliche Intelligenz* 31, 1 (01 Mar 2017), 73–83.

[17] Carl Hewitt. 2010. Actor Model of Computation: Scalable Robust Information Systems. arXiv:1008.1459 [cs.PL]

[18] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[19] Koen V. Hindriks. 2009. Programming Rational Agents in GOAL. In *Multi-agent programming: Languages, platforms and applications*. Chapter 4, 119–157.

[20] C. Kaiser and Jean-François Pradat-Peyre. 2003. Chameneos, a concurrency game for Java, Ada and others. In *ACS/IEEE International Conference on Computer Systems and Applications*. IEEE, 62–70.

[21] Felipe Meneguzzi and Lavindra De Silva. 2015. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review* 30, 1 (2015), 1–44.

[22] Felipe Meneguzzi and Michael Luck. 2009. Norm-based behaviour modification in BDI agents. *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS* 1, 177–184.

[23] Mostafa Mohajeri Parizi, Giovanni Sileno, and Tom van Engers. 2019. Integrating CP-Nets in Reactive BDI Agents. In *PRIMA 2019: Principles and Practice of Multi-Agent Systems*. Springer International Publishing, 305–320.

[24] Anand S. Rao. 1996. AgentSpeak(L): BDI Agents Speak out in a Logical Computable Language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Agents Breaking Away: Agents Breaking Away* (Einhoven, The Netherlands) *(MAAMAW '96)*. Springer-Verlag, Berlin, Heidelberg, 42–55.

[25] Anand S. Rao and Michael P. Georgeff. 1995. BDI agents: From theory to practice.. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS1995)*. 312–319.

[26] Sebastian Rodriguez, Nicolas Gaud, and Stéphane Galland. 2014. SARL: A General-Purpose Agent-Oriented Programming Language. In *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, Vol. 3. 103–110.

[27] Yoav Shoham. 1993. Agent-oriented programming. *Artificial intelligence* 60, 1 (1993), 51–92.

[28] Dhirendra Singh, Sebastian Sardina, Lin Padgham, and Geoff James. 2011. Integrating Learning into a BDI Agent for Environments with Changing Dynamics. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three* (Barcelona, Catalonia, Spain) *(IJCAI'11)*. AAAI Press, 2525–2530.

[29] Simeon Visser, John Thangarajah, James Harland, and Frank Dignum. 2016. Preference-based reasoning in BDI agent systems. *Autonomous Agents and Multi-Agent Systems* 30, 2 (2016), 291–330.