Dynamic logic of questions
Minica, S.A.

Citation for published version (APA):

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: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3
Implementing Questioning Dynamics

In this chapter we present and document the Haskell implementation behind DELq. Some outstanding features of the implementation are the following: modeling the dynamics of issue-epistemic updates via questioning actions, intuitive display of update results, intuitive display of both issue-epistemic models and questioning action models, model checking of issue-epistemic formulae in issue-epistemic structures. The chapter presents and explains the code of one main module `DELQ.lhs` as well as its related functionality contained in auxiliary modules. The code is written in Haskell [55], [68], in the style of Knuth’s ‘literate programming’ and it is related to previous epistemic functionality from DEMo [107], and specific DELq functionality from [71].

3.1 A DEMo-like Implementation for DELQ

In this sections we present the implementation behind the DELq results of Chapter 2. We present and discuss the main modules, other modules containing auxiliary functionality are added in a final section. Further illustrations of how the code is useful can be found already in the next section based on the theory already introduced in the other chapters, they are also referred to throughout the chapter and additional illustrations are added.

The working language for this chapter is Haskell. Haskell is a high level, non-strict, purely-functional programming language named after Haskell B. Curry. The overall design of the implementation has a modular structure, with specific modules capturing theoretical aspects and modeling logical components introduced and discussed so far in previous chapters. The `DELQ.lhs` module is behind the general theoretical background for the logic of questions from Chapter 2, and it also contains functionality to model more complex questioning and resolution actions that include questions that are represented by a cover instead of a partition and answering that captures indistinguishability of previous questions.
Chapter 3. Implementing Questioning Dynamics

The following modules are part of the main `DELQ.lhs` module:

**Syntax** The module that defines the data structures for propositional symbols, agent labels and complex formulae of DELq. These definitions are behind syntax of the epistemic language from Section 2.1.2 of Chapter 2, where their intuitive meaning was introduced and discussed extensively.

**Structures** The module that defines the data structures for the issue-epistemic structures and the various action models and event models as well as their inner structure component by component. Specific functionality, like initialization and naming functions are also included in this module. These definitions are behind Section 2.1.1 of Chapter 2, where their intuitive meaning was introduced and discussed in detail.

**BinaryRel** The module containing the type definition and the main functionalities for binary relations. Binary relations are a central component of the issue-epistemic and action structures, therefore operations on and properties of relations are used for many tasks: defining issue and information partitions, computing fixed points, etc. These definitions are behind Section 2.2.1 of Chapter 2, where their intuitive meaning was introduced and discussed.

**Semantics** The module that contains the functionality linking the formal language introduced in the `Syntax.lhs` module to the issue-epistemic models and their functionality from the `Structures.lhs` module. The main pillar of this link is given by the way in which DELq formulae are interpreted in issue-epistemic structures. This is done by a recursive semantic definition for the concept of truth in a structure. All these correspond to Section 2.1.2 of Chapter 2, where they are intuitively illustrated and minutely explained.

**Upgrade** The module containing functionality related to the model upgrade operations discussed so far. These encode model transformations either by means of resolution and refinement based on taking intersections or by using pre-conditions and taking the product upgrade of issue-epistemic structures and questioning action models. These definitions are behind Section 2.2.1 of Chapter 2, where their intuitive meaning is introduced and discussed.

**Display** Auxiliary module containing functionality used to display epistemic structures themselves and results of various operations involving them such as model-checking of formulae, domain naming, listing of expressible formulae, etc. These have been used in Section 3.2 of Chapter 2.

**Shortcuts** Auxiliary module containing the predefined structures used as examples and illustrations from Section 3.2 of Chapter 2.
3.1.  A DEMo-like Implementation for DEL\textsubscript{Q}

### 3.1.1  The Syntax.hs Module

```haskell
module Syntax where
import List

data Prop = P Int | Q Int | R Int | S Int | N Int deriving (Eq,Ord)
data Nomi = Nomi Int deriving (Eq,Ord)

instance Show Prop where
  show (P 0) = "p";
  show (P i) = "p" ++ show i
  show (Q 0) = "q";
  show (Q i) = "q" ++ show i
  show (R 0) = "r";
  show (R i) = "r" ++ show i
  show (S 0) = "s";
  show (S i) = "s" ++ show i
  show (N 0) = "n";
  show (N i) = "n" ++ show i

data Agent = A Int | B Int | C Int | D Int | E Int deriving (Eq,Ord)

instance Show Agent where
  show (A 0) = "a";
  show (A i) = "a" ++ show i
  show (B 0) = "b";
  show (B i) = "b" ++ show i
  show (C 0) = "c";
  show (C i) = "c" ++ show i
  show (D 0) = "d";
  show (D i) = "d" ++ show i
  show (E 0) = "e";
  show (E i) = "e" ++ show i
```

Basic operations on lists are predefined in Prelude.hs which is the main Haskell module containing the standard language definitions and functions. The list functionality is imported from the List module in line 3.

Next, the data-structure for propositional atoms is defined and its basic functionality is added in lines 5-13. Propositions are going to be symbols possible indexed by an integer, we introduce for this data constructors. We want to be able to test propositional symbols for equality and to compare and order propositional symbols, we make the proposition datatype inherit this properties as they are predefined in the standard Haskell type-system by using the deriving (Eq,Ord) command. A special type of propositional symbols are the nominals, line 6. For conceptual reasons we can introduce a distinct data structure for them. For technical reasons propositions and nominals have to belong to disjoint sets as discussed before. In practice, however, this can also be achieved by simply reserving a special symbol for nominals, in our case N, line 5. We give here both alternatives and we will use the most convenient one in subsequent functions.

Finally, we want to be able to display propositional symbols, we achieve this by making Prop an instance of the predefined Show class and by defining the show function for propositions. In case the index is 0 this is not going to be displayed, otherwise we concatenate the character for the index and display the resulting list of characters. We continue by defining the data-structure for agent labels and by adding its basic functionality in lines 15-22. These is completely analogous to what we did for propositional symbols.
Chapter 3. Implementing Questioning Dynamics

The final code block, lines 23-54, defines the data-structure for complex formulae and adds its basic functionality. These follow the basic definitions introduced in Section 2.1.2 of Chapter 2: we have constructors for nominals and propositional atoms, then we have the basic boolean connectives, taking list of formulas for conjunction and disjunction, and we have constructors for the static modalities, which take both a formula and an agent label as parameters. In addition, we also introduce standard group notions for issues, knowledge and their intersection: conditional common knowledge, common knowledge and distributed knowledge, taking a list of agents as an additional parameter, and general knowledge.

```haskell
data Formq = Top | Prop Prop | Nomi Nomi
            | Neg Formq | Conj [Formq] | Disj [Formq]
            | O Agent Formq | X Agent Formq | K Agent Formq | U Formq
            | CO [Agent] Formq | DO [Agent] Formq | EO Formq
            | CX [Agent] Formq | DX [Agent] Formq | EX Formq
            | CCO [Agent] Formq Formq | CCK [Agent] Formq Formq
            | CCX [Agent] Formq Formq deriving (Eq,Ord)

instance Show Formq where
  show Top = "T"
  show (Prop p) = show p
  show (Nomi i) = show i
  show (Neg f) = '~': show f
  show (Conj fs) = '&': show fs
  show (Disj fs) = 'v': show fs
  show (O agent f) = 'Q': show agent ++ show f
  show (X agent f) = 'R': show agent ++ show f
  show (K agent f) = 'K': show agent ++ show f
  show (U f) = 'U': show f
  show (CO group f) = "CQ" ++ show group ++ show f
  show (DO group f) = "DQ" ++ show group ++ show f
  show (EO f) = "EQ" ++ show f
  show (CX group f) = "CX" ++ show group ++ show f
  show (DX group f) = "DX" ++ show group ++ show f
  show (EX f) = "EX" ++ show f
  show (CK group f) = "CK" ++ show group ++ show f
  show (DK group f) = "DK" ++ show group ++ show f
  show (EK f) = "EK" ++ show f
  show (CCO group f1 f2) = "CCQ" ++ show f1 ++ show group ++ show f2
  show (CCX group f1 f2) = "CCX" ++ show f1 ++ show group ++ show f2
  show (CCK group f1 f2) = "CCK" ++ show f1 ++ show group ++ show f2
```

Analogous to propositions, we define the `show` function used to display formulae in our language, using prefix notation for boolean connectives which are applied to lists of formulae. Further illustrations of how this code works for some frequently used formula instances are included in Section 3.2.
3.1.2 The Structures.lhs Module

The list functionality is imported as explained before. The syntax-related functionality is imported from the previously described Syntax.lhs module in line 4. The content of the Probability module will be explained at a later stage.

```haskell
module Structures
where
import List
import Syntax
import Probability

data EIM state = Eim
    [state]
    [Agent]
    [(Agent,state,state)]
    [(Agent,state,state)]
    [(state,[Prop])]
    [state]
    deriving (Eq,Show)
```

Next, the data-structure for issue-epistemic models is defined in lines 7-14 following the basic definitions introduced and discussed in Section 2.1.1 of Chapter 2. Without going again in minute details we remind the main components and explain how they are implemented. The data constructor for issue-epistemic structures takes the following arguments: a list of states, representing the domain of alternatives or possible worlds; a list of labels representing the agents, as introduced before; two lists of triples containing an agent and two states, encoding the issue relation respectively the uncertainty relations, indexed by the corresponding agent label; a list of pairs containing a state and a list of propositional symbols, representing the valuation function assigning propositions to each possible world; the final component is a list of states encoding the actual situation. We want to be able to test for equality and display EIMs, so we make the EIM datatype inherit this properties as they are predefined in the standard Haskell type-system by using the deriving (Eq,Show) command. Further illustrations of how this code works for some concrete instances of EIMs are included in Section 3.2.

Next we add the functions needed to work with the structures just defined.

The function initMq, lines 16-24, is used to generate issue-epistemic models. It takes a list of agents and a list of propositional atoms as arguments, respectively, and returns a “blisfull ignorance” and “pristine questioning” issue-epistemic model, i.e., each agent has both a universal issue relation and a universal uncertainty relation over all possible combinations of the given propositional atoms. The actual situation remains unspecified, using a list of the entire domain.

```haskell
initMq :: (Num state, Enum state) => [Agent] -> [Prop] -> (EIM state)
initMq ags props = (Eim worlds ags accs eqq val points)
    where
```

The next step in the generation of an issue-epistemic model is to name all its states using the nominal n indexed by integer values for every possible world in the domain. This is done by the named function which takes an arbitrary structures and returns it named, see lines 40-45.

The remaining functions have an auxiliary role as follows: starting from line 26, the powerList function defines recursively a list-analogue of the powerset construction, it is used to generate all propositional combinations. The functions sortL and sortR, line 30, respectively, line 34 give two alternative ways of ordering lists, by comparing first their length and second their content. Both require an ordered type as the predefined list ordering functions are used. Finally, the dom function, line 38, takes an EIM and returns its domain.

The data structure for EIMs and its functionality are very flexible, as they can be extended with minor modeling adaptations to capture action structures.

**Questioning Action Models** The action models representing both questioning and resolution actions are a straightforward adaptation of EIMs. Their data-structures is defined in lines 47-54 following the basic definitions introduced and discussed in Section 2.1.1 of Chapter 2. Again, without going in minute details we remind the main components and explain briefly the implementation design. The
3.1. A DEMo-like Implementation for DELQ

data constructor for questioning action structures (QAMs) takes the following arguments: a list of states, representing the arbitrary epistemic events, which in this context stand for possible answers; a list of labels representing the agents, as before; two lists of triples containing an agent and two states, encoding the issue respectively uncertainty relations, as before; a list of pairs containing an event and a issue-epistemic formula, representing the precondition function assigning formulae encoding conditions for execution to questioning events, this replaces the previous valuation function; the final ingredient is a list of events encoding the real or actual question. We want to be able to test for equality and display QAMs, so we make the EIM datatype inherit this properties as they are predefined in the standard Haskell type-system by using the deriving (Eq,Show) command. Further illustrations of how this code works for some concrete instances of QAMs are included in Section 3.2 of the current chapter.

```haskell
data QM state = Qm [state] [Agent] [(Agent,state,state)] [(Agent,state,state)] [(state,Formq)] [state] deriving (Eq,Show)
```

The data structure for a resolution action, lines 56-63, is defined essentially as the one for questioning actions, with the exception that now the list of answers is replaced by a unique event. However, this structural isomorphism hides the fact that in practice the formulae encoding preconditions for execution have a completely different structure. This is why we have both actions represented by distinct data constructors.

The main arguments in describing and generating questioning action structures are given by a list of ignorant agents a list of knowing, or aware, agents and a list of formulae representing content of binary questions together with a designated formula which stands for the content of the actual questioning action:

```haskell
data RM state = Rm [state] [Agent] [(Agent,state,state)] [(Agent,state,state)] [(state,Formq)] state deriving (Eq,Show)
```

The `initAq` function, lines 65-80, generates a model for indistinguishable yes/no questions as follows: two events are generated for each formula, line 70, corresponding to the affirmative and the negative answers; the corresponding positive and negative preconditions are assigned to the generated events, line 71;
the uncertainty relation is constructed starting from line 72, taking into account
the distinction between the two lists of ignorant and aware agents, answers to
the same question remain undetermined for everyone while answers to different
questions are indistinguishable only for the ignorant agents; the issue relation is
set to the identity for all agents, line 77; and the distinguished set of events is the
binary answer to the real question, line 79. The \texttt{precond} function, lines 82-85, is
an auxiliary function used to retrieve the precondition for an event.

\begin{verbatim}
initAq :: (Num state, Enum state) =>
initAq iags kags propfs quest =
  (Qm events (iags ++ kags) accs eqqs prec answers)
  where
    events = [0..(2* (fromIntegral (length propfs)))-1]
    prec = zip events (propfs ++ (map \(x\to (\text{Neg} x)) propfs))
    accs = [(ag,st1,st2) | ag <- iags, st1 <- events, st2 <- events
       ++(ag,st1,st2) | ag <-kags, st1 <-events, st2 <- events,
       or [(precond prec st1) == (precond prec st2),
       (precond prec st1) == (Neg (precond prec st2)),
       (precond prec st2) == (Neg (precond prec st1))] ]
    eqqs = [(ag,st1,st2) | ag <- (iags ++ kags), st1 <- events,
       st2 <- events, st1==st2]
    answers = let q = quest in
       (map \(x\to (\text{fst} x)) (filter \(x\to((\text{snd} x)==q)) prec
    precond :: (Num state, Enum state) =>
        [(state,Formq)] -> state -> Formq
    precond prec e =
       (map \(x\to (\text{snd} x)) (filter \(x\to((\text{fst} x)==e)) prec))!!0
\end{verbatim}

The \texttt{initAr} function, lines 87-98, generates a model for a resolution action
as follows: a list of events matching the formula-list argument is created, line
92, and execution preconditions are assigned to each event, line 93; as previously
for questioning actions, the pattern based on the distinction between the list
arguments containing ignorant and aware agents is used to generate a universal
and an identity uncertainty relation, respectively, line 94 and a universal issue-
relation, line 96; finally, the last argument determines the actual event, line 97.

\begin{verbatim}
initAr :: (Num state, Enum state) =>
initAr iags kags propfs res =
  (Rm events (iags ++ kags) accs eqqs prec answer)
  where
    events = [0..(1* (fromIntegral (length propfs)))-1]
    prec = zip events propfs
    accs = [(ag,st1,st2) | ag <- iags, st1 <- events, st2 <- events]
        ++ [(ag,st1,st2) | ag <- kags, st1 <- events, st2 <- events, st1==st2]
    eqqs = [(ag,st1,st2) | ag <- (iags++kags), st1 <- events, st2 <- events]
    answer = let q = res in
           (map \(x\to (\text{fst} x)) (filter \(x\to((\text{snd} x)==q)) prec))!!0
\end{verbatim}
Such resolution structures work fine with a product update mechanism for public questions, however, in order to capture more complex resolution actions succeeding private questioning actions more elaborate constructions are needed.

The \texttt{pinitAr} function, lines 100-114, generates a resolution action structure that allows for some answers to remain indistinguishable as follows:

The first two arguments are, as before, list containing ignorant respectively aware agents used to generate uncertainty relations that allow for epistemic gradients, line 107, instead of just complete uncertainty as before, now partial uncertainty can be modeled; the issue-relation is, as before, universal, line 112; a list of events matching the formula-list argument is created, line 105, and execution preconditions are assigned to each event, line 106 as explained previously; finally, the last function argument is used to determine the actual event, in line 113.

\textbf{Complex Questioning} As with resolution actions, the raw binary questioning structures are just a first approximation, their basic utilities can be extended to deal with more complex questioning functionality in an analogous way. We give here the code for both complex propositional questions with mutually disjoint answers and complex cover-based questions with overlapping answer-sets.
Chapter 3. Implementing Questioning Dynamics

The `initPropq` function, lines 115-129, generalizes the previous construction from binary yes/no questions to arbitrary partition-based propositional questions. To capture this additional complexity the function arguments are now a list of ignorant agents, a list of aware agents, just like before, but the remaining parameters are lifted one level of abstraction, we have now a list of lists of formulae, standing for the indistinguishable questioning actions and a list of formulae designed to capture the real or actual question.

The function arguments are assumed for now to be adequate for a partition model of questions, i.e. the formulae are assumed to be mutually incompatible, however they do not have to also be jointly exhaustive, the product update mechanism handles such aspects in a standard way. The process of automatic generation proceeds as follows: events are generated for each formula, after collapsing in line 120 by `foldr` one level of abstraction and concatenating all formulae in one list; the corresponding exclusive preconditions are assigned to the generated events, in line 121 by `zip`-ing the two lists; the uncertainty relation is constructed starting from line 122, taking into account the distinction between ignorant and aware agents, answers to the same question remain undetermined for everyone while answers to different questions are indistinguishable only for the ignorant agents, because in the current setting it is possible that answers appear in multiple questions, the answer-list is also used as an additional identification criterion; the issue relation is set, as before, to the identity for all agents, line 126 capturing the fact that questioning actions refine the issue relation; and the actual set of events is taken to be the distinguished formula list given as the last argument, line 128. Further illustrations of how this code works for some concrete examples are included in Section 3.2 later in this chapter.
eqqs = [(ag, st1, st2) | ag <- (iags ++ kags), st1 <- events, st2 <- events, st1==st2]
answers = [fst33 ((filter (\x -> (and [(snd33 x)==
answ, (trd33 x)==quest]) ) (zip3 events ((foldr (++)
[] frmlist)) (foldr (++) [] (map (\x -> take (length x)
(repeat x)) frmlist))))!!0)]

The initPropCov function, lines 135-155, generalizes the previous construction from partition-based propositional questions to questions based on a cover by eliminating the requirement that answers to questions have to be disjoint and allowing for answers with overlapping extensions.

In order to deal with this additional complexity the initPropCov function, lines 135-155, takes as arguments the lists of ignorant and aware agents, just like before, a list of formulae lists, but the last parameter is now a list-formula pair, representing the real question action and the designed actual answer, respectively. Auxiliary projection functions for triples are defined in lines 131-133.

The process of automatic generation proceeds as follows: events are generated for each formula, again by folding all the answers in a single list, in line 140; the arbitrary preconditions are zip-ed with the generated events, in line 141; the uncertainty relation is constructed using triples containing events, questions and answers starting from line 142, the distinction between ignorant and aware agents has the same role as before, because in the current setting answers may have nonempty intersections and appear in multiple questions, the answer-list is also used as an additional identification criterion; the issue relation is set, as before, to the identity for all agents, in line 150, capturing the fact that questioning actions refine the issue relation by distinct events, even thought some might have overlapping or identical preconditions; and the actual event is constraint now to a singleton list taking into account both the precondition and the question list to which it belongs, line 152. Further illustrations of how this code works for some concrete examples are included in Section 3.2.

Complex Resolution It is now time to extend our models for resolution actions in a similar way to allow a formal structure matching the gradient level of abstraction in the actions with indistinguishable questions discussed so far.

A first step in this direction will be to parametrize resolution by the history of previous questioning actions. This can be done by the use of preconditions. The initAres function, lines 157-177, has the same structure as previous ones but assigns preconditions to event in a more complex way.

The componentwise generation process is the following: events are generated for each formula, by taking binary answers, in line 162; corresponding binary issue preconditions are zip-ed with the generated events, in line 163; the uncertainty relation is constructed by pairing events with affirmative respectively negative issue-precondition, (see Section 3.2 for the intuitive example), starting from line 167, the distinction between ignorant and aware agents has the same role as
explained before; the issue relation is the universal relation for all agents, line 175, capturing the fact that resolution actions do not raise further issues; and the actual event is determined by the last function argument, at line 176.

Resolution actions can in some cases override the structure of the issue relation and give either more or less information than a mere answer to the questions raised so far would allow. There is no reason why such information flow scenarios should be excluded from the formalism. And indeed such situations can be captured by using the constructor defined by the \texttt{initAres2} function, lines 179-192.

The components that are unchanged from the previous constructor are the event set, from line 184, the binary issue-precondition formulae, from line 185, the universal issue relation, from line 190, and the designated actual event, in line 191; the changed component in this new resolution model consists in assigning an universal indistinguishability relation between events to the oblivious agents, starting from line 188, while the aware agents have a transparent access to the content of the resolution action. Further intuitive illustrations of how this code works and some concrete examples of resulting resolution actions and their epistemic effect are included in Section 3.2 later in this chapter.

\begin{verbatim}
initAres :: (Num state, Enum state) =>
initAres iags kags propfs actf =
(Qm events (iags ++ kags) accs eqqs prec resol)
  where
  events = [0..(2* (fromIntegral (length propfs)))-1]
  prec = zip events ((map \(x\) -> (Conj [ x, Disj [ O (iags!!0) x, 
  0 (iags!!0) (Neg x)]]) propfs) ++ (map \(x\) -> (Conj [ 
  Neg x, Disj [ 0 (iags!!0) x, 0 (iags!!0) 
  (Neg x)]]) (propfs)))
  accs = [(ag,st1,st2) | ag <- iags, st1 <- events, st2 <- events, 
  ((elemIndices st1 events)!!0) <= (length propfs)-1, 
  ((elemIndices st2 events)!!0) <= (length propfs)-1]
  ++ [(ag,st1,st2) | ag <- iags, st1 <- events, 
  st2 <- events, ((elemIndices st1 events)!!0) >= 
  (length propfs), ((elemIndices st2 events)!!0) >= 
  (length propfs)] ++ [(ag,st1,st2) | ag <- kags, 
  st1 <- events, st2 <- events, st1==st2 
  eqqs = [(ag,st1,st2) | ag <- (iags ++ kags),st1 <- events,st2 <- events]
  resol = let q = actf in 
  map \(x\) -> (fst x)) (filter \(x\) ->((snd x)==q)) (zip events propfs))

initAres2 :: (Num state, Enum state) =>
initAres2 iags kags propfs actf =
(Qm events (iags ++ kags) accs eqqs prec resol)
  where
  events = [0..(2* (fromIntegral (length propfs)))-1]
  prec = zip events ((map \(x\) -> (Conj [ x, Disj [ O (iags!!0) x, 
  0 (iags!!0) (Neg x)]]) propfs) ++ (map \(x\) -> (Conj [Neg x,Disj 
  [ 0 (iags!!0) x, 0 (iags!!0) (Neg x)]])) (propfs)))
\end{verbatim}
3.1. A DEMo-like Implementation for DELQ

\[
\text{accs} = [(ag, st1, st2) \mid ag <\!\!\!< \text{iags}, st1 <\!\!\!< \text{events}, st2 <\!\!\!< \text{events}] +
\[
[(ag, st1, st2) \mid ag <\!\!\!< \text{kags}, st1 <\!\!\!< \text{events}, st2 <\!\!\!< \text{events}, st1 == st2]
\]
\[
eqqs = [(ag, st1, st2) \mid ag <\!\!\!< \text{iags ++ kags}, st1 <\!\!\!< \text{events}, st2 <\!\!\!< \text{events}]
\]
\[
\text{resol} = \text{let } q = \text{actf in map } (\lambda x \rightarrow (\text{fst } x))
\]
\[
(\text{filter } (\lambda x \rightarrow (\text{snd } x) == q) (\text{zip } \text{events propfs}))
\]

The \text{initAres3} function, lines 194-209, takes resolution actions to the other extreme in this large spectrum of modeling possibilities by allowing resolutions in which all the questions raised so far are publicly resolved for all agents.

The components that are unchanged from the previous constructor are the event set, from line 199, the binary issue-precondition formulae, from line 200, the universal issue relation, from line 207, and the designated actual event, in line 208; the changed component in this new resolution model consists in assigning an identity indistinguishability relation between events to all agents in line 204.

\[
\text{initAres3} :: (\text{Num state}, \text{Enum state}) \Rightarrow
\]
\[
[\text{Agent}] \rightarrow [\text{Agent}] \rightarrow [\text{Formq}] \rightarrow \text{Formq} \rightarrow (\text{QM state})
\]
\[
\text{initAres3 } \text{iags kags propfs actf} =
\]
\[
(\text{Qm events (iags ++ kags) accs eqqs prec resol})
\]
\[
\text{where}
\]
\[
\text{events} = [0..(2* (\text{fromIntegral } (\text{length propfs})))\text{-}1]
\]
\[
\text{prec} = \text{zip events } (\text{map } (\lambda x \rightarrow (\text{Conj } [ x, \text{Disj } [ 0 \text{ (iags!!0) x, 0 \text{ (iags!!0) (Neg x)]}]) \text{ propfs}) ++ (\text{map } (\lambda x \rightarrow (\text{Conj [ Neg x, Disj } [ 0 \text{ (iags!!0) x, 0 \text{ (iags!!0) (Neg x)]}]) \text{ propfs})))}
\]
\[
\text{accs} = [(ag, st1, st2) \mid ag <\!\!\!< \text{iags}, st1 <\!\!\!< \text{events}, st2 <\!\!\!< \text{events}, st1 == st2] ++ [(ag, st1, st2) \mid ag <\!\!\!< \text{kags}, st1 <\!\!\!< \text{events}, st2 <\!\!\!< \text{events}, st1 == st2]
\]
\[
eqqs = [(ag, st1, st2) \mid ag <\!\!\!< \text{iags ++ kags}, st1 <\!\!\!< \text{events}, st2 <\!\!\!< \text{events}]
\]
\[
\text{resol} = \text{let } q = \text{actf in map } (\lambda x \rightarrow (\text{fst } x))
\]
\[
(\text{filter } (\lambda x \rightarrow (\text{snd } x) == q) (\text{zip } \text{events propfs}))
\]

Finally, the \text{initAres4} function, lines 211-229, models a resolution action with an arbitrary gradient of publicity for the answer events. The components that remain unchanged are the event set, from line 216, the binary issue-precondition formulae, from line 217, the universal issue relation, from line 226, and the designated actual event, in line 208; the changed component in this new resolution model consists in assigning an indistinguishability relation that groups the events by division modulo the length of the list of answer events, starting from line 221.

Further intuitive illustrations of how this code works for some concrete resolution actions are discussed in Section 3.2 later in this chapter.

Although all the constructions presented and explained in this section seem to lead to an industrious process, they follow the usual requirements for an exercise in the ‘art of modeling’ for various scenarios occurring in luxuriant practical applications, moreover, the generation of adequate models is automated, indicating the fact that the choice of the right issue-epistemic gradient for a particular context to be modeled does is not given by logic but is merely a contingent choice determined by an adequate model of the situation at hand.
Chapter 3. Implementing Questioning Dynamics

The real advantage of having all these models is that they can be handled in a unitary manner by the product update mechanism, as we will see shortly.

3.1.3 The BinaryRel.lhs Module

Before we go on to discuss the product update mechanism we will spend some time presenting the basic definitions and functionality used in the implementation for binary relations as already introduced in 2.2.1 of Chapter 2.

The module starts by importing the standard list functionality and defining, in line 5 the datatype for binary relations as a list of pairs. Next a containment function for list is defined to be used for comparing relations represented as pair-lists, at line 7, also a function that decides identity for relations, by ignoring the order in which the pairs are present in the list, is defined at line 9.

```haskell
module BinaryRel
where

type Rel a = [(a, a)]

cnv :: Rel a -> Rel a
    cnv r = [ (y, x) | (x, y) <- r |

c containedIn :: Eq a => [a] -> [a] -> Bool
    containedIn xs ys = all (\x -> elem x ys) xs

sameR :: Ord a => Rel a -> Rel a -> Bool
    sameR r s = sort (nub r) == sort (nub s)

cnv :: Rel a -> Rel a
    cnv r = [ (y, x) | (x, y) <- r ]

infixr 5 @@
    (@@) :: Eq a => Rel a -> Rel a -> Rel a
```
3.1. A DEMo-like Implementation for DELQ

The image of an element under a relation is defined in line 25 and the reflexive transitive closure and reflexive closure of a relation are introduced in lines 32 respectively line 36 using the least fix-point definition from line 28. These definitions are going to be useful later on, for instance to define the semantics of the common knowledge modality or for partition refinement.

3.1.4 The Semantics.lhs Module

The Semantics.lhs module uses the functionality presented so far, starting from line 3, to give a recursive definition for truth of DELQ formulae in issue-epistemic structures as introduced previously in Section 2.1.2 of Chapter 2.
Chapter 3. Implementing Questioning Dynamics

models :: Ord state => EIM state -> state -> Formq -> Bool
models m w Top = True
models m w (Prop p) =
  elem p (concat [props|(w',props) <- val, w'=w ])
models m w (Neg f) = not (models m w f)
models m w (Conj fs) = and (map (models m w) fs)
models m w (Disj fs) = or (map (models m w) fs)
models m@(Eim _ _ eqq _ _) w (O agt f) =
  and (map (flip (models m) f) (rightS (relq agt m) w))
models m@(Eim _ _ acc _ _ _) w (X agt f) =
  and (map (flip (models m) f) (rightS (relx agt m) w))
models m@(Eim _ _ _ _ _ _) w (K agt f) =
  and (map (flip (models m) f) (rightS (relk agt m) w))
models m@(Eim dom ags _ eqq _ _) w (U f) =
  and (map (flip (models m) f ) dom)

relk :: Agent -> EIM a -> Rel a
relk a m@(Eim _ _ _ _ _ _) = [ (x,y) | (agent,x,y) <-acc, a == agent]
relq :: Agent -> EIM a -> Rel a
relq a m@(Eim _ _ acc _ _ _) = [ (x,y) | (agent,x,y) <-eqq, a == agent]
relx :: (Eq a) => Agent -> EIM a -> Rel a
relx a m = intersect (relk a m) (relq a m)

In line 8, the auxiliary function takes an issue-epistemic structure and returns its domain. Before defining the usual local definition of truth at a state or from a pointed perspective, we introduce the same notion at the level of a set of worlds from line 11. This notion turns out to be often useful, especially in game contexts where players are uncertain about the real situation.

The most important function in this module is the models function introduced in lines 14-28. It takes as input an issue-epistemic structure, a state in its domain and a formula and returns a boolean value. The definitions proceed along the expected lines starting with the boolean cases, for which the valuation of the model provides the needed information, and continuing to formulae containing more complex modalities, for which the functionality related to the issue and epistemic relations becomes crucial. The relevant information about the issues, epistemic uncertainty and their interrelation for an agent or even groups of agents is contained inside the model received as a function parameter. In order to access this information and use it in recursive truth value computations, three additional auxiliary functions are defined starting from lines 33, 30 and 36, respectively.

3.1.5 The Upgrade.lhs Module

We have now all the requested ingredients to proceed towards implementing the functionality needed for the central dynamic actions for DELq as discussed in Section 2.2.1 of Chapter 2. In this section we will show that the product update
3.1. A DEMo-like Implementation for DELQ

can be used as a unifying formal mechanism encompassing the luxuriant variety
of questioning actions that can be modelled by DELq. We will also show how
using the issue relation on top of the traditional epistemic relation can extend
the product update mechanism in an interesting and useful way.

The module starts again in the standard way by incrementally importing some
needed functionality, from modules previously explained at lines 3-7.

```haskell
module Upgrade
where
import List
import Syntax
import Structures
import Semantics
import Shortcuts
```

Next, the `upgradeq` function, lines 9-27, implements the product update mech-
anism between issue structures and models of questioning actions using, as before,
the componentwise construction of the models’ constituents described before.

The new domain consists of pairs of worlds and events with matching pre-
conditions, from line 15; the two models that are the function’s arguments are
assumed to take the same list of agents, which is also used in the output model,
line 17; the new uncertainty and issue relations are constructed by combining
the old ones in both the issue-epistemic model and the action model, see lines
18 respectively 21; the new valuation conserves the previous one for the world in
the pair, line 24; the new actual world is uniquely determined by the event with
matching preconditions in the real question list, line 26.

```haskell
upgradeq:: (Eq state, Ord state) =>
EIM state -> QM state -> EIM (state, state)
upgradeq m@(Eim dom agts accs eqqs val act)
q@(Qm evs ags acs eqs prec answ) =
(Eim udom uagts uaccs ueqqs uval uact)
where
  udom = [(w,e) | w <- dom, e <- evs,
            (models m w (precondition q e))]
  uagts = [x | x <- agts]
  uaccs = [(ag,(w,e),(v,f)) | ag <- ags, (w,e) <- udom,
            (v,f) <- udom, (ag,w,v) 'elem' accs,
            (ag,e,f) 'elem' acs]
  ueqqs = [(ag,(w,e),(v,f)) | ag <- ags, (w,e) <- udom,
            (v,f) <- udom, (ag,w,v) 'elem' eqqs,
            (ag,e,f) 'elem' eqs]
  uval = [(w,e),l) | (w,e) <- udom, l <- (map (snd x) val),
            ((valuation m w) == l)]
  uact = [(w,e) | w <- act, e <- answ,
            (models m w (precondition q e))]
```

The functions `valuation`, line 29, and `precondition`, line 33, have an auxil-
ary role, the first takes an EIM and a state and returns its valuation, the second
takes a question model and an event and returns its precondition for execution.

```haskell
valuation :: Eq state => EIM state -> state -> [Prop]
valuation m@(Eim dom agts accs eqqs val act) w =
    (snd ((filter (\x -> ((fst x) == w)) val)!!0)

precondition :: Eq state => QM state -> state -> Formq
precondition q@(Qm evs ags acs eqs prec answ) e =
    (snd ((filter (\x -> ((fst x) == e)) prec)!!0)
```

The product update mechanism for resolution will be applied on two sample models: an issue-epistemic model, line 37, and a resolution model, line 40.

```haskell
eimsample = (upgradeq (initMq [a,b] [Syntax.P 0,Q 0])
    (initAq [a] [b] [p,q] p))
resample = (initAres [a] [b] [p,q] p)
```

Now we have all the required ingredients to introduce an implementation of a product rule that uses both equivalence relations to model resolution actions. The first and most obvious way in which to make good use of the expressive power introduced by the fact that our models use two relations is by considering their intersection in defining the action’s effects. The `exclam` function given in lines 42-46 does exactly this. It implements the resolution by intersection dynamic action defined before. This is a very simple and already useful mechanism that makes the interdependence between questions and information explicit.

```haskell
exclam :: (Eq state, Ord state) => EIM state -> EIM state
exclam m@(Eim dom agts accs eqqs val act) =
    (Eim dom agts accsr eqqs val act)
    where
        accsr = accs `intersect` eqqs
```

A more complex resolution action, more suitable for realistic scenarios involving the various gradients of publicity in a resolution discussed before will extend the product update mechanism by allowing a combination between the two relations to be the driving force in the product upgrade mechanism.

```haskell
upgraderes :: (Eq state, Ord state) =>
    EIM state -> QM state -> EIM (state, state)
upgraderes m@(Eim dom agts accs eqqs val act)
    mq@(Qm evs ags acs eqs prec answ) = (Eim udom uagts uaccs ueqqs uval uact)
    where
        udom = [(w,e) | w <- dom, e <- evs,
            (models m w (precondition mq e))]
        uagts = [x | x <- agts]
        uaccs = [(ag,(w,e),(v,f)) |
            ag <- uagts, (w,e) <- udom,
            (v,f) <- udom,
            (ag,w,v) `elem` accs, or
            (ag,v,w) `elem` eqqs,(ag,e,f) `elem` acs]
        ueqqs = [(ag,(w,e),(v,f)) |
            ag <- uagts, (w,e) <- udom,
            (v,f) <- udom,
            (ag,w,v) `elem` eqqs,
            (ag,e,f) `elem` eqs]
```
3.1. A DEMo-like Implementation for DELQ

\[ uval = \left\{ (w,e) \mid (w,e) \in \text{udom}, l \in \text{nub} \left( \text{map} (\lambda x \rightarrow (\text{snd} x)) \text{val} \right), (\text{valuation} m w) = l \right\} \]

\[ uact = \left\{ (w) \mid w \in \text{act}, e \in \text{answ}, (\text{models} m w (\text{precondition} mq e)) \right\} \]

\[ \text{recondition} :: \text{Eq state} \Rightarrow \text{RM state} \rightarrow \text{state} \rightarrow \text{Formq} \]

\[ \text{recondition} r@(\text{Rm evs ags acs eqs prec answ}) e = \]

\[ (\text{snd} (\text{filter} (\lambda x \rightarrow ((\text{fst} x) == e)) \text{prec}!!0)) \]

The upgrade function implements this aspects in lines 48-63 by the following componentwise construction: the new domain consists of pairs of worlds and events with matching preconditions, line 53; the function’s arguments are an issue-epistemic model and a resolution model assumed to take the same list of agents, which is also used to construct the output model, line 54; the new uncertainty relation is constructed by a disjunctive combination of old uncertainty between states and previous issue equivalence between events, from line 55; the new issue relations are constructed in the same way as before from their correspondents in both the given resolution model and the issue-epistemic model, line 58; the new valuation conserves the valuation of the world in the pair, line 61; the new actual world is determined by the event with matching preconditions in the real question list, line 63. The last function, from line 65, has an auxiliary role, it takes a resolution model and an event and returns its precondition for execution.

3.1.6 The DELQ.lhs Module

The DELQ.lhs module puts together all the functionality presented so far in an unitary framework. It starts by importing the functionality in the corresponding modules as discussed previously. Then it includes some additional functions.

\[ \text{extension} :: (\text{Ord} a) \Rightarrow \text{Formq} \rightarrow \text{EIM a} \rightarrow [a] \]

\[ \text{extension} f m = \text{filter} (\lambda x \rightarrow (\text{models} m x f)) (\text{dom} m) \]
Chapter 3. Implementing Questioning Dynamics

boxk :: (Eq b) => [b] -> Agent -> EIM b -> [b]
boxk s a m = nub (map snd (filter (\x -> (fst x) 'elem' s) (relk a m)))

boxk_minus :: (Eq b) => [b] -> Agent -> EIM b -> [b]
boxk_minus s a m = nub (map fst (filter (\x -> (snd x) 'elem' s) (relk a m)))

The last function in the block, line 24, returns the set of states that satisfy a formula via the intersection modality, making crucial use of nominals.

intersOp f m = filter (\x -> models m x (k_ a
                  (Conj [f, (\_ a ((noml m1 [x]!!0)))])) (dom m1)

nominals :: (Eq a) => EIM a -> [Formq]
nominals m@(Eim _ _ _ _ val _) = map (\x -> (Prop x)) (map (\x -> (x!!((length x)-1))) (map snd val))

noml :: (Eq a) => EIM a -> [a] -> [Formq]
noml m l = map (\x -> (Prop x)) ((map (\x -> (x!!((length x)-1)))) (map snd val))

valuationL :: EIM a -> [(a, [Prop])]
valuationL m@(Eim _ _ _ _ val _) = val

forms :: (Ord state) => EIM state -> [Formq ]
forms m = map (\y -> Disj y) (map (\x -> noml m x) (powerList (dom m)))

relk_a :: Agent -> QM state -> Rel state
relk_a a m@(Qm _ _ acc _ _ _) = [(x,y) | (agent,x,y) <- acc, a == agent ]

relq_a :: Agent -> QM state -> Rel state
relq_a a m@(Qm _ _ eqq _ _) = [(x,y) | (agent,x,y) <- eqq, a == agent]

Next, there are two functions dealing with generating nominals for an issue-epistemic structure given as input, line 27. This is useful in order to deal with intersection modality and to allow for full expressibility when describing a named model. The second function generates the nominal list for a given subset of the domain in an issue-epistemic structure, which is useful to generate all the expressible formulae given a model and a hybrid language, line 31.

The valuation of a model can be retrieved by the function given in line 36, and can be used to generate all the expressible formulae as disjunctions of nominals in the powerset of the domain, as described starting from line 39.

Finally, two more auxiliary functions are given at line 43 and line 47, they take a questioning action model and an agent and return the relation pairs indexed by that agent, for the epistemic and the issue relations, respectively.
Further intuitive illustrations of how this code works for some concrete resolution actions are discussed in Section 3.2 of the current chapter.

This concludes the exposition of the core DELq functionality contained in the most important modules implementing theoretical aspects. The remaining ones have supporting role with only practical and applicative importance.

### 3.2 Illustrations using the Implementation

In this chapter we will discuss some general consequences of the DELq approach explained so far. We will support and give concreteness to our discussion by using illustrative intuitive examples, for this purpose we will make use of code output for both ease of exposition and computational precision.

In order to do this we will guide the reader through the bare minimum needed to grasp the examples discussed. We do this by linking some of the concrete examples already discussed and illustrated in the text with their representation as both standard tuple-notation and implementation generated output.

Epistemic Issue Structures (EIMs) can be automatically generated by \texttt{DELQ.hs} in a standard way. For facilitating the current presentation we refer the reader to our previous intuitive diagramatic representation in Figure 2.12. The starting EIM in our Example 2.4.5 is generated and represented as follows:

\begin{verbatim}
*QPR> dpq (initMq [a,b] [Syntax.P 0,Q 0])
[0,1,2,3]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
(a,[[0,1,2,3]])
(b,[[0,1,2,3]])
[3]
(a,[[0,1,2,3]])
(b,[[0,1,2,3]])
\end{verbatim}

Intuitively, this says that we have two agents \(a, b\) ignorant about two facts \(p, q\) and with a universal issue relation on the domain, the world in which both facts are true represents the actual situation. This also corresponds to the following structure in standard tuple-notation \(M = \langle W, a \approx b, a \sim b, V \rangle\): \(W = \{0, 1, 2, 3\}\), \(V = \{p \mapsto \{1, 3\}, q \mapsto \{2, 3\}\}\), \(a \approx b = \{0, 1, 2, 3\} \times \{0, 1, 2, 3\} = a \sim b\).

In a similar fashion, the Action Issue Model (AIM) from Example 2.4.5 can be automatically generated by \texttt{DELQ.hs} in a standard way. The second model in Figure 2.12 is generated and represented in the implementation as follows:

\begin{verbatim}
*QPR> dpa (initAq [a] [b] [p,q] p)
[0,1,2,3]
[(0,[p]),(1,[q]),(2,~p),(3,~q)]
(a,[[0,1,2,3]])
(b,[[0,2],[1,3]])
[0,2]
(a,[[0],[1],[2],[3]])
(b,[[0],[1],[2],[3]])
\end{verbatim}
Intuitively, we have here two agents $a, b$ and four events corresponding to the possible ways to answers the two questions $p?$ and $q?$. Each of the events has the corresponding propositional atoms and their negation. Later on, in order to capture more interesting situation such preconditions can become more complex formulae in our epistemic issue language designed to capture questioning content and its relation with agents’ information. This also corresponds to the following structure represented in standard tuple-notation $Q = \langle E, a \approx b, a \sim b \rangle$:

$$E = \{0, 1, 2, 3\}, \quad P = \{0 \mapsto p, 1 \mapsto q, 2 \mapsto \neg p, 3 \mapsto \neg q\}, \quad a \approx b = \{0, 1, 2, 3\} \times \{0, 1, 2, 3\}, \quad a \sim b = \{0, 1, 2, 3\} \times \{0, 1, 2, 3\} = Id(E), \quad a \approx b = \{0, 2\} \times \{0, 2\} \cup \{1, 3\} \times \{1, 3\}.$$

Here is a good place to introduce the basics about how issue-epistemic formulae are generated, used, and represented in Haskell. Formulas can be visualized in `DELQ.hs` in a compact way, we give here some illustrative examples covering most frequently used formula instances:

<table>
<thead>
<tr>
<th>DELQ.hs</th>
<th>Formula</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg (Conj [0 a p, Neg p])</td>
<td>$\neg (Q_a p \land \neg p)$</td>
<td>$\neg &amp;[Qap, \neg p]$</td>
</tr>
<tr>
<td>Disj [Neg (X a (Prop(P 0))), p]</td>
<td>$R_a p \rightarrow p$</td>
<td>$R[p]$</td>
</tr>
<tr>
<td>Disj [K a p, K a (Neg p)]</td>
<td>$K_a p \lor \hat{K}_a \neg p$</td>
<td>$K[p, \neg Kp]$</td>
</tr>
<tr>
<td>U (Disj [0 b q, 0 b (Neg q)])</td>
<td>$U(Q_b q \lor Q_b \neg q)$</td>
<td>$U[Qbp, Qb]$</td>
</tr>
<tr>
<td>U (Conj [Prop(N 1), X a (K b p)])</td>
<td>$U(n_1 \land R_a K_b p)$</td>
<td>$U[n1, RaKbp]$</td>
</tr>
</tbody>
</table>

Continuing with our correspondence between the diagramatic representation in Figure 2.12 and the code output, we can see that a is ignorant with regard to what question was asked, we model this by a universal indistinguishability relation on the set of events (answers). The second agent $b$, on the other hand, is aware of what the content of the questioning action is. The identity issue relation on the domain of events captures the fact that both agents are aware about possible answers to the questioning actions. Finally, the actual question is $p?$, we represent this by a list of distinguished events in the domain. The set of distinguished events is not a singleton, as in the standard DEL action structure, but the set of possible answers to the real question $\{p, \neg p\}$.

**Discussion (Determinism)** One important assumption in the standard DEL modeling using product update was the fact that both epistemic structures and action models are deterministic. This background assumption was present in the formalism in the uniqueness of the real world and of the actual event. Some agents are ignorant about the real world and some agents cannot determine what epistemic action has taken place. But this kind of nondeterminism is merely a reflection of epistemic uncertainty not genuine lack of determinism in the real situation modeled. The fact that the perfectly informed sl modeler of the epistemic scenario can chose beforehand a designated world and a unique action that really took place reflects precisely the essential deterministic character of the approach.
3.2. Illustrations using the Implementation

One natural question at this point is: Don’t we have to give up determinism when we want to build an adequate model of questioning actions? A quick answer at this stage is: Not yet. Note that EIMs still have a unique world, and we only introduced multiple events in AIMs. However, we still require our models to be deterministic, even though at a higher level. We only allow for a unique questioning action, which can have multiple, not yet determined, answers corresponding to a partition cell. Besides being conceptually interesting, this aspect will be important from a technical point of view during the completeness proof. For this reason, we give it a distinct statement already at this stage:

3.2.1. Definition. [Deterministic AIMs] A questioning structure is a Kripke frame over a set of agents and events together with a set of events corresponding to multiple (but mutually exclusive) answers to a unique designated question.

Using Definition 3.2.1 one can show that the class of EIMs structures is closed under product update with deterministic AIMs and partition questions.

We now return to our exposition of Example 2.4.5. The resulting updated model can now be computed in our implementation of DELq using standard product update; the result is visualized by DELLQ.hs in the following way:

```
*QPR> dpq (reornamexq (upgradeq
     (initMq [a,b] [Syntax.P 0,Q 0]) (initAq [a] [b] [p,q] p)))

[0,1,2,3,4,5,6,7]
[(0,[]),(1,[]),(2,[p]),(3,[p]),(4,[q]),(5,[q]),(6,[p,q]),(7,[p,q])]
(a,[[0,1,2,3,4,5,6,7]])
(b,[[0,2,5,6],[1,3,4,7]])
[6]
(a,[[0,5],[1,3],[2,6],[4,7]])
(b,[[0,5],[1,3],[2,6],[4,7]])
```

The resulting model has a unique actual world and agents have the expected issue relation capturing the natural intuitions, namely, both agents have an issue relation that captures the possible answers but only b can distinguish between the two possibilities, while a is uncertain which of the alternative is the case.

Starting from this first simple illustration, we will proceed towards a more general logical framework in which examples like this can be captured formally. The next step in the standard DEL methodology is to define simultaneously a language that uses modalities over such AIM structures and the structures in which the event-preconditions are formulated in a language describing them.

Let L be a set of atomic sentences closed under boolean operators, epistemic, issue and intersection modalities for a set of agent labels and $\gamma ? \varphi$ where $\gamma$ is a deterministic AIM and $\varphi \in L$. The following step in the DEL methodology is to lift the indistinguishability relation from events to action structures. For DEL this was a simple translation from uncertainty between events in a structure with the same domain to an epistemic relation between structures themselves.
Chapter 3. Implementing Questioning Dynamics

Because we have now both epistemic and questioning equivalence relations the task at hand requires more work. Another conceptual difficulty that prevents applying the same strategy comes from the fact that we are already using a reduction. Our AIM structures are meant to capture questioning actions while the domain of an AIM contains not questions but answers to questions. However, we will show that a natural lifting that captures the basic intuitions and keeps the product method unchanged is still possible. For this purpose we will use a standard ordering of questions as sets of answers. This design option has many natural formal properties and also captures the intended intuitions:

3.2.2. Definition. [Lifting] Let \( \gamma = \langle E, \approx, \sim, P, Q \rangle \) and \( \gamma' = \langle E', \approx', \sim', P', Q' \rangle \) be two AIMS, then:

\[
\gamma' \approx \gamma \quad \text{iff} \quad E = E', P = P' \text{ and } \forall e \in Q, e' \in Q' : e \approx e'
\]

\[
\gamma' \sim \gamma \quad \text{iff} \quad E = E', P = P' \text{ and } \forall e \in Q, e' \in Q' : e \sim e'
\]

We have now all the ingredients for proving the soundness of the axioms given in Section 2.4.3 using the semantics already introduced there. The crucial step used Definition 3.2.1 to restrict the class of structures described.

3.2.1. Proof (Theorem 2.4.8). Let \( \gamma = \langle E, \approx, \sim, P, Q \rangle \) be an AIM. Fix a pair \( \langle W, w \rangle \). As \( \gamma \) is a yes/no question, it follows that \( W \models_w P(e) \) for some \( e \in Q \). Using Definition 3.2.1 we also know that there is a unique such \( (w, e) \). Pick the \( e \) disjunct in the rhs of the A&P axiom. Assume that \( \langle W, w \rangle \otimes \gamma \models Q_u \varphi \). Take some \( \gamma' \) such that \( \gamma \sim \gamma' \). By Definition 3.2.2, this \( \gamma' \) is of the form \( \langle E, \approx', \sim', P', Q' \rangle \) for some \( Q' \) such that \( \forall e \in Q, e' \in Q' : e \sim e' \). Let \( w \approx w' \). We have two cases: \( Q = Q' \), and \( Q \neq Q' \). In both cases we have \( e \approx e' \) iff \( e = e' \). Then \( (w', e') \) is a world of \( \langle W, w \rangle \otimes \gamma \), and indeed \( (w, e) \approx (w', e') \). Since \( Q \) is a unique set of yes/no answers we have to consider two possibilities for \( e \in Q \): if \( \langle W, w' \rangle \models \neg \text{pre}(e) \) then \( \langle W, w' \rangle \models \text{pre}(e) \rightarrow [\gamma'] \varphi \) trivially and we are done, otherwise, \( \langle W, w' \rangle \models \text{pre}(e) \) and we can use the previous assumption. From \( \langle W, w \rangle \otimes \gamma \models Q_u \varphi \) we get that \( \langle W \otimes \gamma, (w', e') \rangle \models \varphi \). This means that \( \langle W, w' \rangle \otimes \gamma' \models \varphi \). Hence \( \langle W, w' \rangle \models \text{pre}(e) \rightarrow [\gamma'] \varphi \). Since \( \gamma' \) and \( w' \) were arbitrary, \( \langle W, w \rangle \models \Lambda e \approx e \models [\gamma'] \varphi \). The other direction is similar.

Pick the \( e \) disjunct in the rhs of the A&K axiom. Assume that \( \langle W, w \rangle \otimes \gamma \models K_a \varphi \). Take some \( \gamma' \) such that \( \gamma \sim \gamma' \). By Definition 3.2.2, this \( \gamma' \) is of the form \( \langle E, \approx', \sim', P, Q' \rangle \) for some \( Q' \) such that \( \forall e \in Q, e' \in Q' : e \sim e' \). Let \( w \approx w' \). We have two cases: \( Q = Q' \), and \( Q \neq Q' \). In both cases we have \( e \sim e' \). Then \( (w', e') \) is a world of \( \langle W, w \rangle \otimes \gamma \), and indeed \( (w, e) \sim (w', e') \). Now our assumption that \( \langle W, w \rangle \otimes \gamma \models K_a \varphi \) implies that \( \langle W \otimes \gamma, (w', e') \rangle \models \varphi \). This means that \( \langle W, w' \rangle \otimes \gamma \models \varphi \). Hence \( \langle W, w' \rangle \models [\gamma] \varphi \). Since \( \gamma' \) and \( w' \) were arbitrary, \( \langle W, w \rangle \models \Lambda e \approx e \models [\gamma] \varphi \). The other direction is similar.
3.2. Illustrations using the Implementation

The rest of the proof continues along the lines of the standard DEL methodology described in the previous section with the only caveat that now we need to define a complexity measure for formulae containing questioning action modalities by taking the complexity of the preconditions.

We also get the axioms for resolution action for free since we used the most convenient formal model that gives the semantic meaning by taking the previously discussed intersection. Even so, formal convenience does not always match conceptual clarity, we still have an unfulfilled modeling desideratum on our list.

\[
*QPR> dpq (exclam (renameq (upgradeq
  (initMq [a,b] [Syntax.P 0,Q 0]) (initAq [a] [b] [p,q] p)))
)[0,1,2,3,4,5,6,7]
[0, [1], (1, []), (2, [p]), (3, [p]), (4, [q]), (5, [q]), (6, [p,q]), (7, [p,q])]
(a, [[0,5], [1,3], [2,6], [4,7]],)
(b, [[0,5], [1,3], [2,6], [4,7]])
[6]
(a, [[0,5], [1,3], [2,6], [4,7]])
(b, [[0,5], [1,3], [2,6], [4,7]])
\]

Consider the effects of taking the undifferentiated epistemic effects of the intersection resolution action in our previous questioning product update model which produce the result represented in the previous code output.

A brief inspection of the resulting model and a basic analysis of its intuitive meaning reveals that it is fair to say that it would be expected that a resolution action can also be modulated epistemically. There can be levels of underlying dynamics: some agents can be aware of the fact that a resolution action happens but still be opaque about its content, while for others the content is transparent.

In our example, since \( a \) was unaware which of the questions was asked, hearing an affirmative answer should not have the same epistemic effect as a transparent answer, or public announcement. Nor should hearing the negative resolution to one of two undistinguished questions have the same effect as receiving an negative answer to any of them. Our rough intersection semantics misses such subtleties.

Once again, we can use the framework of product update to capture the desired epistemic effects. In order to do this we will make crucial use of the gain in expressive power coming from having a language capable of talking about both facts, epistemic realities, questioning conditions and their mutual interdependence. Again we will proceed first by means of an illustrative example:

\[
*QPR> dpq (renameq (upgradeq
  (initMq [a,b] [Syntax.P 0,Q 0]) (initAq [a] [b] [p,q] p)))
)[0,1,2,3,4,5,6,7]
[0, [1], (1, []), (2, [p]), (3, [p]), (4, [q]), (5, [q]), (6, [p,q]), (7, [p,q])]
(a, [[0,2,5,6], [1,3,4,7]],)
(b, [[0,2,5,6], [1,3,4,7]])
[6]
(a, [[0,5], [1,3], [2,6], [4,7]])
(b, [[0,5], [1,3], [2,6], [4,7]])
\]
Chapter 3. Implementing Questioning Dynamics

The main task is to come up with a formal structure representing a resolution action which captures the natural intuitions about the epistemic evolution. Such a model can be again automatically generated by the implementation in a standard way. The structure we need here is represented by \texttt{DELQ.hs} as follows:

```haskell
dpa (initAres [a] [b] [p,q] p) 
[0,1,2,3]
[(0,\&[p,v[Qap,Qa^p]]), (1,\&[q,v[Qaq,Qa^q]])
 (2,\&[\neg p,v[Qap,Qa^p]]), (3,\&[\neg q,v[Qaq,Qa^q]])
 (a,[[0,1],[2,3]])
 (b,[[0],[1],[2],[3]])
 [0]
 (a,[[0,1,2,3]])
 (b,[[0,1,2,3]])
```

A new feature in comparison to AIMs for questioning actions is the more complex preconditions for resolution, they have now, in addition to the factual component, a questioning aspect designed to capture the conditions on the structure of the issue relation. Next, the resolution does not change the issue structure by raising new questions, therefore both agents have a universal relation. Finally, agent \textit{a} can only distinguish between affirmative and negative resolution, like when hearing an opaque ‘Yes’ answer in a conversation that could refer to any of two previous questions. We have used agent-dependent epistemic preconditions before, the issue-related content of the preconditions for resolution goes beyond only modeling epistemic effects and factual aspects. In order to capture the intended meaning we add a further generalization to the product update rule. This can now go beyond using only indistinguishability and it is natural to take full advantage of both equivalence relations. The new uncertainty can be now derived from two alternative and complementary sources: both previous uncertainty and issue-indistinguishability. Both aspects and their mutual dependence are needed to capture the dynamics of joint observation and prediction uncertainty.

The rule that makes sense in this context is the following:

\[(w,e) \sim (v,f) \iff w \sim v \text{ and } (w \approx v \text{ or } e \sim f)\]

This rule allows a finer grained modeling of the dependence between two sources of uncertainty in an epistemic scenario, and captures expected intuitions:
The new kind of resolution only partially resolves agent \( a \)'s prediction uncertainty due to his previous observational limitation about the question asked. He can distinguish the affirmative answers from the negative ones but acquires no further knowledge about the situation. The flow of information is modulated by two distinct sources: previous uncertainty and current observational limitations.

Further gradations in the level of assertive or questioning content implicitly contained in a resolution action can now be spelled out using the same underlying formalism. However, capturing this variation is not a logical task. The underlying logic is given in all cases by the same extended product rule, while the remaining enterprise consists merely in designing the adequate action structures for the concrete resolution scenario to be captured. Witness the following situations:

\[
\begin{align*}
*\text{QPR} &> \text{dpa} \ (\text{initAres2} \ [a] \ [b] \ [p,q] \ p) \\
&([0,1,2,3]) \\
&((0,\&[p,vQap,Qa\neg p]),(1,\&vQaq,Qa\neg q)), \\
&(2,\&vQap,Qa\neg p),(3,\&vQaq,Qa\neg q)) \\
&(a,[[0,1,2,3]]) \\
&(b,[[0],[1],[2],[3]]) \\
&[0] \\
&(a,[[0,1,2,3]]) \\
&(b,[[0,1,2,3]])
\end{align*}
\]

\[
\begin{align*}
*\text{QPR} &> \text{dpa} \ (\text{initAres4} \ [a] \ [b] \ [p,q] \ p) \\
&([0,1,2,3]) \\
&((0,\&[p,vQap,Qa\neg p]),(1,\&vQaq,Qa\neg q)), \\
&(2,\&vQap,Qa\neg p),(3,\&vQaq,Qa\neg q)) \\
&(a,[[0,2],[1,3]]) \\
&(b,[[0],[1],[2],[3]]) \\
&[0] \\
&(a,[[0,1,2,3]]) \\
&(b,[[0,1,2,3]])
\end{align*}
\]

\[
\begin{align*}
*\text{QPR} &> \text{dpa} \ (\text{initAres3} \ [a] \ [b] \ [p,q] \ p) \\
&([0,1,2,3]) \\
&((0,\&[p,vQap,Qa\neg p]),(1,\&vQaq,Qa\neg q)), \\
&(2,\&vQap,Qa\neg p),(3,\&vQaq,Qa\neg q)) \\
&(a,[[0],[1],[2],[3]]) \\
&(b,[[0],[1],[2],[3]]) \\
&[0] \\
&(a,[[0,1,2,3]]) \\
&(b,[[0,1,2,3]])
\end{align*}
\]

The following code outputs are resulting models capturing the intuitively adequate gradients in information flow predicted by our extended product rule:

\[
\begin{align*}
*\text{QPR} &> \text{dpq} \ (\text{reornamq} \ (\text{upgraders} \ (\text{reornamq} \ (\text{upgradeq} \\
&\ (\text{initMq} \ [a,b] \ [\text{Syntax.P 0,Q 0}] \ (\text{initAq} \ [a] \ [b] \ [p,q] \ p))))) \\
&\ (\text{initAres2} \ [a] \ [b] \ [p,q] \ p)) \\
&([0,1,2,3,4,5,6,7]) \\
&((0,[]),(1,[2]),(2,[p]),(3,[p]),(4,[q]),(5,[q]),(6,[p,q]),(7,[p,q])) \\
&(a,[[0,1,2,3,4,5,6,7]])
\end{align*}
\]
The same gradation can be applied to the issue relation, our initial questioning action model was one extreme in a large spectrum of intermediate possibilities of mixing informative and questioning content in various levels. This aspect also opens the possibility of further comparing the DEL approach to alternative approaches of questioning phenomena and to unravel some implicit modeling assumptions in an explicit formal representation.

Another interesting aspect is that the new resolution rule gives a formal way of capturing the widespread phenomenon that much information, even if both truthful and known, might still not be available for announcement. This gives a formal justification for using procedural restrictions in both games with epistemic moves and protocols for inquiry and communication.

So far we have used only Yes/No questions. Doesn’t this choice drastically limit our modeling options? The quick answer at this stage is: Not yet. The framework can be extended without any alteration to more complex propositional questions by merely lifting the construction from formulae to lists/sets of formulae. The following code outputs illustrate exactly such situations.

Once again, the extension from binary questions to arbitrary propositional questions falls under the same logical framework using product update. The only extra needed work is an empirical exercise in designing the right AIM for the given scenario to be modeled. This is a very interesting exercise in the ‘art of modeling’ without significant conceptual or computational consequences.
The crucial fact here is the requirement that the designed AIM is an adequate one. If it respects the postulate of determinacy and uses appropriate sets of mutually inconsistent answers, then all the formal details can be adapted without ado, and all results transfer in corresponding reformulations.

**Discussion: Nondeterminism** What about modeling questions that do not have mutually inconsistent answers? In other words, does the DEL product upgrade mechanism still work when we switch from a classical model of questions modeled as partitions to one which has a cover as the underlying formal structure? We will discuss the conceptual implications of this change and we will show that the formal product mechanism still works under minor adaptations.

So we are still working with a unique actual world in our initial EIMs, but now we take a question to be an arbitrary set of formulae seen as subsets of the domain of possibilities, including ones with a nonempty intersection.

Therefore, our previous AIM structures in which we take as the designated set of actual events all the answers to the actual question can generate multiple actual worlds in the model obtained after performing product update.

It can be argued that this is meaningful as a representation of nondeterminism at a subjective level in many scenarios of social interaction, or even in an objective sense in contexts of dependence between measurements of entangled states in quantum physics. However, we identify it as an open problem whether the formal
details needed in the proofs can be adapted for a non-deterministic setting, and reserve a study of valid principles for nondeterminism for a future occasion.

Our next step will be to illustrate how our previous mechanism works if we change the questioning structures to ones having both a set of events representing the real question and a designated event for the actual answer. As in the standard DEL approach, and as we did before for issue relations, we use preconditions for action execution to reduce multiple consistent answers to questioning actions with unique effects. This is a good place to illustrate how model checking of formulae involving epistemic and issue effects is performed in our Haskell implementation.

\[\text{*DELQ}\> \text{dpq m1} \]
\[[0,1,2,3] \quad [(0,[n]),(1,[p,n1]),(2,[p,n2]),(3,[q,n3])] \]
\[[0,1,2,3] \quad (a,[[0,1,3],[2]]) \quad (a,[[0,2,3],[1]]) \]

The truth-value of formulas can be computed at a state in an epistemic-issue model. The following is an illustration of how \text{DELQ.hs} performs model checking in the issue-epistemic model \text{m1} represented in the code output above:

\[\text{*DELQ}\> \text{models m1 0 (impl (x_ a p) (Conj[o_ a p,k_ a p])}) \]
\[True \]
\[\text{*DELQ}\> \text{models m1 0 (dimpl (x_ a p) (Conj[o_ a p,k_ a p])}) \]
\[False \]
\[\text{*DELQ}\> \text{models m1 0 (impl (x_ a q) (Conj[o_ a q,k_ a q])}) \]
\[True \]
\[\text{*DELQ}\> \text{models m1 0 (dimpl (x_ a (Conj[n3,q])))} \]
\[\text{Conj[o_ a (Conj[n3,q]),k_ a (Conj[n3,q])])} \]
\[True \]
\[\text{*DELQ}\> \text{models m1 0 (Conj)} \]
\[\text{[o_ a (Conj[n2,p]), k_ a (Conj[n1,p])])} \]
\[True \]
\[\text{*DELQ}\> \text{models m1 0 (Conj)} \]
\[\text{[o_ a (Conj[n2,p]), k_ a (Conj[n1,p]), x_ a p}) \]
\[False \]

Now, we can attach preconditions for mutually consistent answers to distinct events in our action models and we obtain the expected results. The only necessary adjustment is to use a singleton set of real answers in the questioning model during the product computation. In this way the class of deterministic EIMs remains closed under product update. Here is how this can be done for a disjunctive question that can receive three overlapping answers:

\[\text{*QPR}\> \text{dpa (initPropCov [a] [b]} \]
\[ [[\text{Disj[p,q]},p,q],[\text{Disj[p,Neg q]},p,Neg q]] \quad [\text{Disj[p,q],p,q}] \quad p) \]
\[[0,1,2,3,4,5] \quad [(0,v[p,q]),(1,p),(2,q),(3,v[p,q]),(4,p),(5,q)] \quad (a,[[0,1,2,3,4,5]]) \quad (b,[[0,1,2],[3,4,5]]) \]
This setting can also be used to obtain completeness results about long term inquiry procedures. The discussion so far about adapting the proof techniques from informative actions to questioning actions is equally relevant in a long term protocols setting. We give here the proof for the basic case with binary questions.

3.2.2. Proof (Lemma 2.6.8). By structural induction on $\varphi \in \mathcal{L}_{DELQ}$. Base case: straightforward. Boolean cases: straightforward. We start from the right to left direction: For each $h \in \mathcal{H}_{\text{can}}$:

$$\text{if } \mathcal{H}_{\text{can}}, h \models \varphi \text{ then } \varphi \in \lambda(h)$$

We proceed by considering case by case the most relevant situations:

1. Question/Issue Modality Case:

   (a) Questioning Action Case:

   We start by recalling some relevant valid principles in this context, that we will use later on:

   Axiom 1 (Protocol Asking and Negation):

   $$\langle \gamma? \rangle \neg \varphi \leftrightarrow \langle \gamma? \rangle \top \land \neg \langle \gamma? \rangle \varphi$$

   Axiom 2 (Protocol Asking and Partition):

   $$\langle \gamma? \rangle Q_i \varphi \leftrightarrow \langle \gamma? \rangle \top \land ((\gamma \land Q_i (\gamma \rightarrow \langle \gamma? \rangle \varphi)) \lor (\neg \gamma \land Q_i (\neg \gamma \rightarrow \langle \gamma? \rangle \varphi)))$$

   Take an arbitrary history $h \in \mathcal{H}_{\text{can}}$. Assume that $Q_i \psi \not\in \lambda(h)$ (1). For simplicity let $h = w\gamma$ with $w \in W_0$ and $\gamma \in \mathcal{L}_{EL}$. The argument can easily be generalized to deal with the general case along the lines of the argument below.
Chapter 3. Implementing Questioning Dynamics

By $\lambda$ being a MCS: (1) $\Rightarrow \neg Q_i \psi \in \lambda(h)$ (2). By the construction of legal histories in Definition 2.6.6 we have: (2) $\Rightarrow \langle \gamma?\rangle \neg Q_i \psi \in \lambda(w)(3).$

By Axiom 1 above: (3) $\Rightarrow \neg(\gamma?) Q_i \psi \in \lambda(w)(4)$ By Axiom 2 above and using de Morgan equations we have: (4) $\Rightarrow \neg(\gamma? \wedge Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi)) \vee \neg(\gamma \wedge Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi)) \in \lambda(w)$ (5)

By Construction: $\langle \gamma?\rangle \top \in \lambda(w)$ (6) By Disjunctive Syllogism: (5) & (6) $\Rightarrow \neg((\gamma \wedge Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi)) \vee (\gamma \wedge Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi))) \in \lambda(w)$ (7) By de Morgan equivalence: (7) $\Rightarrow \neg(\gamma \wedge Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi)) \wedge \neg(\gamma \wedge Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi)) \in \lambda(w)$ (8) By de Morgan again we have: (8) $\Rightarrow (\gamma \vee \neg Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi)) \wedge (\gamma \vee \neg Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi)) \in \lambda(w)$ And we can rewrite this as:

$$(-\gamma \wedge \neg Q_i(\neg \gamma \Rightarrow \langle \gamma?\rangle \psi)) \vee$$

$$(-\gamma \wedge \gamma) \vee (-Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi) \wedge \gamma) \vee$$

$$(-Q_i(\gamma \Rightarrow \langle \gamma?\rangle \psi) \wedge \neg Q_i(\neg \gamma \Rightarrow \langle \gamma?\rangle \psi)) \in \lambda(w)$$

We proceed by considering the disjuncts case by case:

Case 1: The $\gamma \wedge \neg \gamma \in \lambda(w)$ case is ruled out by Lemma 2.6.7.

Case 2: If we have $\neg \gamma \wedge \neg Q_i(\neg \gamma \Rightarrow \langle \gamma?\rangle \psi) \in \lambda(w)$ then we can make the sub-claim:

The following is a consistent set:

$$v_0 = \{\theta \mid Q_i \theta \in \lambda(w)\} \cup \{\neg(\neg \gamma \Rightarrow \langle \gamma?\rangle \psi)\}$$

Suppose not. Then there are formulae: $\theta_1, \ldots, \theta_m$ such that:

$$\vdash \bigwedge_{j=1}^{m} \theta_j \Rightarrow (\neg \gamma \Rightarrow \langle \gamma?\rangle \psi)(12)$$

and for $j = 1, \ldots, m, Q_i \theta_j \in \lambda(w)$. Then, by standard modal reasoning (Necessitation and Distribution of Q) we have:

$$(12) \Rightarrow \vdash \bigwedge_{j=1}^{m} Q_i \theta_j \Rightarrow Q_i(\neg \gamma \Rightarrow \langle \gamma?\rangle \psi)(13)$$

Hence, we also have: (13) $\Rightarrow Q_i(\neg \gamma \Rightarrow \langle \gamma?\rangle \psi) \in \lambda(w)(14)$

Because $\lambda$ is a MCS: (14) contradicts (Case 2).

By Lindenbaum’s Lemma any consistent set can be extended to a MCS. Hence there exist a MCS $v$ such that $v_0 \subseteq v$ (9). Therefore, by construction we have: (9) $\Rightarrow w \sim_1^0 v$ and $w \gamma? \sim_1^\text{can} v \gamma?$(10) Hence $\Rightarrow \langle \gamma?\rangle \top \in \lambda(v)$ and $\neg(\gamma?) \psi \in \lambda(v)(11)$ and by the Axiom 1: (11) $\Rightarrow \langle \gamma?\rangle \neg \psi \in \lambda(v)$. By LH construction: (11) $\Rightarrow \neg \psi \in \lambda(v\gamma)?(12)$, hence (12) $\Rightarrow \psi \notin \lambda(v\neg \gamma?)(13)$.

By induction hypothesis: (13) $\Rightarrow H_{\text{can}}, v\psi? \not\models \psi$ (14) and by standard modal semantics: (14) $\Rightarrow H_{\text{can}}, w\gamma? \not\models Q_i\psi$, as desired.
Case 3: If we have \((-Q_i(\gamma \rightarrow \langle \gamma ? \rangle \psi) \land \gamma) \in \lambda(w)\) then we can reason completely analogously to case 2.

Case 4: If we have \((-Q_i(\gamma \rightarrow \langle \gamma ? \rangle \psi) \land -Q_i(-\gamma \rightarrow \langle \gamma ? \rangle \psi)) \in \lambda(w)\) we can use case 2 and case 3 to reach the desired conclusion.

(b) Resolution Action Case:
We start by recalling some relevant valid principles in this context:

Axiom 3 (Protocol Resolution and Negation):

\[
(\!\!\neg \phi \leftrightarrow (\!\!\top \land \neg (\!\!\phi)
\]

Axiom 4 (Protocol Resolution and Partition):

\[
(\!\!Q_i \phi \leftrightarrow (\!\!\top \land Q_i(\!\!\phi)
\]

This case is a situation of a commutating behavior so we are done.

2. Intersection Modality Case: Behaves completely similar to the already explained Question/Issue modality for both cases (a) and (b) below:

(a) Questioning Action Case and
(b) Resolution Action Case.

3. Epistemic/Knowledge Modality Case:

(a) Questioning Action Case:
We start by recalling some relevant valid principles in this context:

Axiom 1 (Protocol Asking and Negation):

\[
(\gamma ? \neg \phi \leftrightarrow (\gamma ?) \top \land \neg (\gamma ?) \phi
\]

Axiom 6 (Protocol Asking and Knowledge):

\[
(\gamma ?) K_i \phi \leftrightarrow (\gamma ?) \top \land K_i(\gamma ?) \phi
\]

This case is a situation of a commutating behavior so we are done.

(b) Resolution Action Case:
We start by recalling some relevant valid principles in this context, that we will use later on:

Axiom 3 (Protocol Resolution and Negation):

\[
(\!\!\neg \phi \leftrightarrow (\!\!\top \land \neg (\!\!\phi)
\]

Axiom 7 (Protocol Resolution and Knowledge):

\[
(\!\!K_i \phi \leftrightarrow (\!\!\top \land R_i(\!\!\phi)
\]
This is an interesting combination as it relies essentially on the intersection modality so we will give it closer attention.

Take an arbitrary history $h \in H^{\text{can}}$. Assume that $K_i \psi \not\in \lambda(h)$ (1). For simplicity let $h = w!$ with $w \in W_0$ and $!$ a resolution symbol. The argument can easily be generalized to deal with the general case along the lines of the argument below.

By $\lambda$ being a MCS: (1) $\Rightarrow \neg K_i \psi \in \lambda(h)$ (2). By the construction of legal histories in Definition 2.6.6 we have: (2) $\Rightarrow \langle ! \rangle \neg K_i \psi \in \lambda(w)$ (3). Using de Morgan:

$\neg \langle ! \rangle \top \lor \neg R_i \langle ! \rangle \psi \in \lambda(w)$ (5).

By Construction: $\langle ! \rangle \top \in \lambda(w)$ (6) By Disjunctive Syllogism: (5) and (6) $\Rightarrow \neg R_i \langle ! \rangle \psi \in \lambda(w)$ (7). We make the next sub-claim:

The following set is consistent:

$$v_0 = \{ \theta \mid R_i \theta \in \lambda(w) \} \cup \{ \neg \langle ! \rangle \psi \}$$

Suppose not. Then there are formulae: $\theta_1, \ldots, \theta_m$ such that:

$$\vdash \bigwedge_{j=1}^{m} \theta_j \rightarrow \neg \langle ! \rangle \psi$$

and for $j = 1, \ldots, m, R_i \theta_j \in \lambda(w)$. Then, by standard modal reasoning (Necessitation and Distribution of Q) we have:

$$(12) \Rightarrow \vdash \bigwedge_{j=1}^{m} R_i \theta_j \rightarrow R_i \neg \langle ! \rangle \psi$$

Hence, we also have: (13) $\Rightarrow R_i \neg \langle ! \rangle \psi \in \lambda(w)$ (14)

Because $\lambda$ is a MCS: (14) contradicts the initial assumption (1).

By Lindenbaum’s Lemma any consistent set can be extended to a MCS. Hence there exist a MCS $v$ such that $v_0 \subseteq v$ (9). Therefore, by construction we have: (9) $\Rightarrow w! \sim_i^{\text{can}} \cap \approx_i^{\text{can}} v!$ and from this it follows by construction that $w! \sim_i^{\text{can}} \cap \approx_i^{\text{can}} v!$ (10). Hence $\Rightarrow \langle ! \rangle \top \in \lambda(v)$ and $\neg \langle ! \rangle \psi \in \lambda(v)$ (11) and by Axiom 3: (11) $\Rightarrow \langle ! \rangle \neg \psi \in \lambda(v)$. By LH construction: (11) $\Rightarrow \neg \psi \in \lambda(v!)$ (12), hence (12) $\Rightarrow \psi \not\in \lambda(v!)$ (13).

By induction hypothesis: (13) $\Rightarrow H^{\text{can}}, w! \not\models \psi$ (14) and by standard modal semantics: (14) $\Rightarrow H^{\text{can}}, w! \not\models R_i \psi$ and $K_i \psi$, as desired.

We continue now with the remaining converse direction. Base case: straightforward. Boolean cases: straightforward. We will consider the interesting cases involving modalities in more detail next.
3.2. Illustrations using the Implementation

1. Epistemic/Knowledge Modality Case:

(a) Resolution Action Case: Let us start by recalling some relevant valid principles in this context, that will be useful later in the proof:

Axiom 7 (Protocol Resolution and Knowledge):

\[ ⟨⟨!⟩⟩K_i ϕ \leftrightarrow ⟨⟨!⟩⟩⊤ ∧ R_i(⟨⟨!⟩⟩ϕ) \]

Take an arbitrary history \( h ∈ H^{can} \) such that \( h = w! \). The argument can easily be generalized to deal with the general case \( h = w!_1 \cdots !_n \), along the lines of the argument below.

Assume that: \( K_i ψ ∈ λ(h) \) (1). Suppose: \( h' ∈ H^{can} \) and \( h \sim_i h' \) (2). By CM construction: (2) ⇒ \( h' = v! \), for some \( v ∈ H_0 \) with \( w \sim_i^0 v \). By LH construction: (1) ⇒ \( ⟨⟨!⟩⟩K_i ψ ∈ λ(w) \) (4). By the previously introduced Axiom 7: (4) ⇒ \( ⟨⟨!⟩⟩⊤ ∧ R_i(⟨⟨!⟩⟩ϕ) ∈ λ(w) = w \) (5). By CM construction: (2) ⇒ \( w \sim_i^0 v \) (6). Using the modal semantics of \( R_i \): (5) and (6) ⇒ \( ⟨⟨!⟩⟩ψ ∈ v = λ(v) \) (7). Using the LH construction we have: (7) ⇒ \( ψ ∈ λ(v!) = λ(h') \) (8). From the Induction Hypothesis it follows that: \( H^{can}, h' \models ψ \). Therefore, by Modal Semantics we get \( H, h \models K_i ψ \) as desired.

(b) Questioning Action Case: Let us start by recalling some relevant valid principles in this context: Axiom 6 (Protocol Asking and Knowledge):

\[ ⟨⟨ϕ?⟩⟩K_i ψ \leftrightarrow ⟨⟨ϕ?⟩⟩⊤ ∧ K_i⟨⟨ϕ?⟩⟩ψ \]

This case is a situation of a commutating behavior so we are done.

2. Intersection Modality Case:

(a) Questioning Action Case:

Let us start by recalling some relevant valid principles in this context, that will be useful later on during the proof:

Axiom 8 (Protocol Asking and Intersection):

\[ ⟨⟨γ?⟩⟩R_i ϕ \leftrightarrow ⟨⟨γ?⟩⟩⊤ ∧ ((γ ∧ R_i(γ → ⟨⟨γ?⟩⟩ϕ)) ∨ (¬γ ∧ R_i(¬γ → ⟨⟨γ?⟩⟩ϕ))) \]

Take an arbitrary history \( h ∈ H^{can} \) such that \( h = wγ \). The argument can be generalized to deal with the general case \( h = wγ_1 \cdots γ_n \). along the lines of the argument below. This generalization essentially relies on having a unique choice of the relevant disjunct in every component in the long term history consisting of a finite sequence of questioning actions. The availability of such a unique choice is ensured by Definition 3.2.1.
Chapter 3. Implementing Questioning Dynamics

Assume that: $R_i\psi \in \lambda(h)$ (1). Suppose that we have: $h' \in H^\text{can}$ and $h \sim^0_i \cap \approx^0_i h'$ (2). By CM construction: (2) $\Rightarrow h' = v\gamma?$, for some $v \in H_0$ such that $w \sim^0_i \cap \approx^0_i v$. From the LH construction it follows that: (1) $\Rightarrow \langle \gamma? \rangle R_i\psi \in \lambda(w)$ (4). By the previously introduced Axiom 8: (4) $\Rightarrow (\langle \gamma? \rangle \top \land ((\gamma \land R_i(\gamma \rightarrow \langle \gamma? \rangle \psi)) \lor (-\gamma \land R_i(-\gamma \rightarrow \langle \gamma? \rangle \psi))) \in \lambda(w) = w$. Hence we have either $(\gamma \land R_i(\gamma \rightarrow \langle \gamma? \rangle \psi)) \in \lambda(w) = w$ or $(-\gamma \land R_i(-\gamma \rightarrow \langle \gamma? \rangle \psi)) \in \lambda(w) = w$. All we have to do at this stage is to pick the right disjunct, because of Definition 3.2.1 we know there is such a disjunct and the result of the choice function is well defined. Because $\lambda$ is a MCS either one or the other is the case. Here $\gamma$ is a yes/no question but the argument can also be extended along the same lines to arbitrary partition-based propositional questions.

Assume wlog: $(-\gamma \land R_i(-\gamma \rightarrow \langle \gamma? \rangle \psi)) \in \lambda(w) = w$ (5). By CM construction: (2) $\Rightarrow w \sim^0_i \cap \approx^0_i v$ (6). Using the modal semantics of $R_i$: (5) and (6) $\Rightarrow \langle \gamma? \rangle \psi \in v = \lambda(v)$ (7). Using the LH construction we have: (7) $\Rightarrow \psi \in \lambda(v\gamma?) = \lambda(h')$ (8). From the Induction Hypothesis it follows that: $H^\text{can}, h' \models \psi$. Therefore, by Modal Semantics we get $H, h \models R_i\psi$ as desired.

(b) Resolution Action Case:

Let us start by recalling some relevant valid principles in this context:

Axiom 6 (Protocol Resolving and Intersection):

$$\langle ! \rangle R_i\phi \leftrightarrow \langle ! \rangle \top \land R_i(\langle ! \rangle \phi)$$

This case is a situation of a commutating behavior so we are done.

3. Issue/Question Modality Case: Behaves completely similar to the already explained Intersection modality for both cases of (a) Questioning Action and (b) Resolution Action.

This concludes the proof. □