Reasoning about Belief in Social Software using Modal Logic

Faculty of Humanities
Department of Philosophy
Cognitive Artificial Intelligence

Thesis for the Degree of Bachelor of Science

February 2010

Author: *Ronald de Haan* Supervisor: *prof. dr. Jan van Eijck*

Contents

1 Introduction **4**
- 1.1 Social Software 4
- 1.2 Plausibility-based belief 4
- 1.3 Belief in Social Software 4
- 1.4 Objectives 5
- 1.5 Relevance 5
- 1.6 Structure 5

2 Motivation **6**
- 2.1 Bankruptcy 6

3 The Logic **6**
- 3.1 The Static Logic 6
  - 3.1.1 Constraints 8
- 3.2 Abbreviations 9
- 3.3 Updates 9
  - 3.3.1 Constraints 12
- 3.4 Distinction 13

4 An Analysis of Bankruptcy **13**
- 4.1 Bankruptcy game 13
- 4.2 An algorithm 18
- 4.3 A Logical Approach 19
- 4.4 Turning the tide 24
  - 4.4.1 Reversing belief 24
  - 4.4.2 Money guarantee 25
  - 4.4.3 Real world correspondence 26
- 4.5 Advantages and disadvantages 26

5 Soundness and Completeness of the Static Logic **27**

6 Well-behavedness of the Updates **35**
- 6.1 Preservation of Properties 35
- 6.2 Reduction of Updates to Static Logic 37
- 6.3 Bisimulation 39

7 Model Checking Software **42**
- 7.1 Possible Improvements 51

8 Conclusion **51**

8.1 Further Research 52

A Model Checking Software Source **56**

A.1 SetOrd 56

A.2 Rel 58

A.3 ModelChecker 60

A.4 Updates 66

A.5 Display 74

Abstract

Social software is the interdisciplinary research program in which social procedures are analyzed and designed using formal, mathematical methods. The analysis of certain procedures requires explicit mention of belief. We develop a logic, based on propositional dynamic logic, that allows us to explicitly reason about belief in social software. This logic consists of a static logic, and dynamic updates. We show how this logic can be used by analyzing the example in which a bank can go bankrupt only because of clients' beliefs. We also give a complete axiomatization of the static logic, and we show (analogously to [vBvEK] and [vEW]) how the updates can be reduced to the static logic. Moreover, we implement model checking software for our logic. Finally, we give suggestions as to what purposes our logic can be used as well.

1 Introduction

1.1 Social Software

Social software is an umbrella term for an interdisciplinary research program that uses mathematical tools from computer science and game theory to analyze social procedures. Social procedures are analyzed as if they were computer algorithms. The name has been coined by Rohit Parikh who was the first to give an analysis of several social procedures in [P_3]. Important notions in the field are concepts such as pareto-optimality, envy-freeness, equity and correctness of procedures. Many different logics can be used to analyze social procedures. In [PP] Pacuit and Parikh give a survey of (logical) tools used in studying social software.

1.2 Plausibility-based belief

Modal logic has been used to model knowledge and belief. Knowledge has been successfully implemented using epistemic logics that handle knowledge updates well. Implementing belief updates has been less straightforward. Because beliefs aren't necessarily true, updates can introduce inconsistencies. A fruitful approach that has been used to model beliefs and belief changes uses plausibility relations. Plausibility relations indicate what certain agents consider more plausible to be true, and certain notions of belief (and knowledge) can be defined in terms of plausibility relations. This approach handles belief updates well. Certain well-behaved logics based on propositional dynamic logic (PDL) have been developed for this purpose in [vBvEK], [vE_2] and [vEW].

1.3 Belief in Social Software

Many situations in social software in which agents are to perform certain actions can be modeled game-theoretically as extensive-form games. Reasoning about actions and results in such game-like situations can be done using propositional dynamic logic. In this use of PDL, programs correspond to actions that agents can perform.

Quite some work has been done on epistemic reasoning in social software. However, in order to analyze certain situations explicit reasoning about belief is needed, and considering knowledge is not enough. Certain logics have been developed to reason about belief using propositional dynamic logic. In this use of PDL, programs correspond to agents' plausibility relations.

In order to reason about both actions and belief in a situation, and their interaction, we combine these two uses of PDL. This results in a PDL-like logic with two kinds of programs:

one kind corresponds to actions that agents can perform and another kind that corresponds to agents' plausibility relations.

1.4 Objectives

We will develop a logic, based on PDL, that allows us to reason about belief in game-like situations in which several agents can choose to perform certain actions. We will use this logic to analyze a situation in which belief plays an important role. We will also prove certain formal properties of this logic.

1.5 Relevance

One of the purposes of Artificial Intelligence is to study human intelligence. One of the ways to get a better understanding of human intelligence is to study human behavior. Social software investigates human behavior in social procedures. Developing better tools that can be used in social software is therefore very useful in our research in the field of Artificial Intelligence.

1.6 Structure

In section 2 we will introduce a situation that illustrates the need for analyzing belief in social software. We will briefly sketch the situation and the aspects of the situation that we want to analyze.

In section 3 we will present a logic that allows us to analyze belief in social software. We define the syntax and semantics of our logic. We then define a notion of updates on our models, and add an update modality to our language. We also discuss several properties of our models and logic.

In section 4 we will give an analysis of the situation we introduced in 2. We show how to model the situation with our semantics and show how to construct this model from scratch using updates. For the analysis we firstly develop an algorithm that goes beyond our models. Then we implement this algorithm using our models and logic. We also show how the situation can be changed using certain updates.

In section 5 we give an axiomatization for the fragment of our logic without updates and we prove completeness of this axiomatization. In section 6 we show that updates are well-behaved by proving certain properties of the update models, including the fact that our logic containing updates reduces to the fragment of our logic without updates.

In section 7 we present a model checking tool for our logic.

In section 8 we will give a brief summary of the logic, the analyses and our findings and draw a number of conclusions. We also give suggestions for further research on this topic.

2 Motivation

In certain procedures or social situations analyzing what agents know is not enough. It may also be relevant to analyze what agents belief. An excellent example of such a situation is the event of a bank going bankrupt (mainly) because people belief it will go bankrupt. A situation that seems to be a news topic every so many years.

2.1 Bankruptcy

Consider the following situation. There is a bank with a number of clients. Every client i can all choose to perform one of two actions: have confidence (i.e., keep their money in the bank), denoted c, or get scared (i.e., take out their savings), denoted s. The bank will go bankrupt if a majority of clients performs s. It is commonly known that every client prefers performing c when the majority of clients does so. It is also commonly known that every client prefers performing s when the majority of clients performs s. Also, every client prefers a majority of people performing c over a majority performing s. Thus, every client would prefer that everyone perform c. However, if any client k believes that a majority believes that the majority is going to perform s, k would perform s as well. In order to analyze this situation formally, we need to have a means of talking about beliefs in such situations.

3 The Logic

We will use a logic consisting of a static and a dynamic part in our analysis of social software. We will use the static part to describe a situation the way it is at a certain moment. The dynamic part (in which we will use product update) will be used to describe an external change of the situation.

3.1 The Static Logic

In order to analyze beliefs and actions formally, we introduce the following logic.

Definition 3.1. An action plausibility model $\mathbf{M}$ (or for short: model) for a set of agents Ag and a set of basic propositions $Prop$ is a tuple $\langle W, P, R, C, V \rangle$ where W is a non-empty set of worlds, P is a function that maps each agent i to a relation P_i (the plausibility relation), R is a set of relations on W (the action relations), C is a function that maps each world in W to an element in $Ag \cup \{\star\}$ (this coloring is intended as the 'control' of a world; $\star$ is used for terminal worlds), and V is a map from W to $\mathcal{P}(Prop)$ (a map that assigns to each world a $Prop$-valuation).

A distinctive action plausibility model is a pair consisting of an action plausibility model and a dinstinguished world in that model.

Definition 3.2. P_i is the plausibility relations for i, where $w \rightarrow^{P_i} w'$ means that w' is at least as plausible as w. From this relation P_i we can extract a relation P_i^s, where $w \rightarrow^{P_i^s} w'$ means that w' is strictly more plausible than w. P_i^s can be obtained from P_i by the following definition: $P_i^s = \{(x, y) \mid x \rightarrow^{P_i^*} y \wedge \neg(y \rightarrow^{P_i^*} x)\}$, where P_i^* is the reflexive, transitive closure of P_i.

Definition 3.3. Formulas φ and programs π (beliefs) and α (actions) are defined inductively as follows. Let q range over $Prop$, p over $\{P_i \mid i \in Ag\} \cup \{P_i^s \mid i \in Ag\}$, i over $Ag \cup \{\star\}$ and a over R.

$$\varphi ::= q \mid c_i \mid \bot \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid [\pi]\varphi \mid [\alpha]\varphi$$

$$\pi ::= p \mid \pi_1; \pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^* \mid \pi^{\smile} \mid ?\varphi$$

$$\alpha ::= a \mid \alpha_1; \alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha^* \mid ?\varphi$$

Abbreviations such as $\varphi_1 \wedge \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$, $\langle\pi\rangle\varphi$ and $\langle\alpha\rangle\varphi$ are defined as usual.

Definition 3.4. Given an action plausibility model $\mathcal{M}$, worlds w, w' in this model and relations π and α, we define the truth of a formula φ inductively.

1. $\mathcal{M}, w \models q$ iff $q \in V(w)$
2. $\mathcal{M}, w \models c_i$ iff $C(w) = i$
3. $\mathcal{M}, w \models \varphi_1 \vee \varphi_2$ iff $\mathcal{M}, w \models \varphi_1$ or $\mathcal{M}, w \models \varphi_2$
4. $\mathcal{M}, w \models \neg\varphi$ iff $\mathcal{M}, w \not\models \varphi$
5. $\mathcal{M}, w \models [\pi]\varphi$ iff in all worlds w' such that $w \rightarrow^{\pi} w'$, $\mathcal{M}, w' \models \varphi$
6. $\mathcal{M}, w \models [\alpha]\varphi$ iff in all worlds w' such that $w \rightarrow^{\alpha} w'$, $\mathcal{M}, w' \models \varphi$

Definition 3.5. We say that a formula φ is valid on a model $\mathcal{M}$ (we write $\mathcal{M} \models \varphi$) iff it is true in all worlds in $\mathcal{M}$ (i.e. $\mathcal{M}, w \models \varphi$ for all $w \in W$).

Definition 3.6. Given a basic relation p we can define binary relations π on W in a model $\mathcal{M}$ inductively.

1. $\pi_1; \pi_2 = \{(x, y) \mid \exists z \in W,\ (x, z) \in \pi_1, (z, y) \in \pi_2\}$ (concatenation)
2. $\pi_1 \cup \pi_2 = \{(x, y) \mid (x, y) \in \pi_1 \vee (x, y) \in \pi_2\}$ (union)
3. $\pi^* = \bigcup_{i=0}^{\infty} \pi^i$ (transitive, reflexive closure)
4. $\pi^{\smile} = \{(x, y) \mid (y, x) \in \pi\}$ (converse)
5. $?\varphi = \{(x, x) \mid \mathcal{M}, x \models \varphi\}$ (test)

With π^0 we denote the relation $?\top$ and with π^{n+1} we denote the relation $\pi^n; \pi$.

Definition 3.7. We can define binary relations α on W in a model $\mathcal{M}$, given a basic relation a. We define rules for concatenation, union, transitive reflexive closure and test analogously to the rules for relations π.

Definition 3.8. We use the abbreviation t for the union of all relations in R. We let the abbreviation τ denote $[t]\bot$. This formula holds in all and only in terminal worlds.

3.1.1 Constraints

We lay several constraints on our models. These constraints formalize several intuitions about belief in social situations. We rule out situations (i.e., models) that conflict with these intuitions.

One such constraint is that all relations P_i are reflexive. Thus, all agents consider any world at least as plausible as itself.

Another constraint is that agents can distinguish worlds that they control from worlds that they do not control. Formally, for all $w, w' \in W$ if $C(w) \neq C(w')$, then $w \not\to^{P_{C(w)}} w'$.

Another such constraint is the constraint of awareness; i.e. agents are aware what actions they can perform. Formally, for all worlds $w, w' \in W$ and every relation $a \in R$ the following holds: if $C(w) = C(w')$, $w \to^{\sim_{C(w)}} w'$ and $\exists w'' \in W$ such that $w \to^a w''$, then $\exists w''' \in W$ such that $w' \to^a w'''$. Note that $\sim_i$ is an abbreviation. See section 3.2 below for more details.

Fourthly, we add a constraint of nondeterminism. We have left open the possibility of

nondeterministic actions[1]. Certain actions can lead from one world to different other worlds. What we want to state now is that an agent cannot distinguish the different outcomes of a nondeterministic action he performs. Formally, $\forall w, w', w'' \in W$, $\forall a \in R$, if $w \rightarrow^a w'$ and $w \rightarrow^a w''$, then $w' \rightarrow^{\sim_{C(w)}} w''$.

3.2 Abbreviations

We can use certain abbreviations for different kinds of epistemic and doxastic relations in our models. The abbreviation for knowledge is taken from [vEW]. Similarly, abbreviations for safe belief, conditional belief and plain belief from [vEW] can be used as well. We will not need those in our analysis, however, so we omit these abbreviations here.

For **knowledge**, we let $\sim_i$ abbreviate $(P_i \cup P_i^{\smile})^*$. According to this definition, every world that is more plausible or less plausible than this world is possible. Knowledge is thus represented by an equivalence relation.

For **unsafe belief**, we let $>_i$ denote P_i^s. For an agent to unsafely belief something in a world w, it must be true in every world that is strictly more plausible than w. If an agent unsafely beliefs φ in a world w, φ doesn't necessarily have to be true in w. The converse of $>_i$ can be abbreviated as $<_i$.

For **weak belief**, we let $>_i^{max}$ abbreviate $P_i^*; ?\neg\langle >_i \rangle \top$. This is a relation from a world to the set of maximally plausible worlds (that are reachable from that world). Something is believed weakly if it is true in every most plausible world. This is a weaker notion of belief than safe, unsafe and plain belief.

For an agent to weakly belief φ it does not have to be true. This notion of weak belief, which allows agents to belief false propositions, leads to interesting possibilities, as we will see in section 4.1. Weak belief of certain propositions can lead to agents performing actions that they would not perform if they knew the actual truth value of these propositions.

3.3 Updates

In order to describe how situations can change, we define a notion of updates. We will define *update models* that describe how the situation changes. These update models are analogous to the update models defined in [vBvEK] and [vEW].

Definition 3.9. Given a language $\mathcal{L}$, $\mathcal{L}$ substitutions are functions of type $\mathcal{L} \rightarrow \mathcal{L}$ that distribute over all language constructs, and that map all but a finite number of basic

[1] A good example of such nondeterministic actions in the domain of social software is an investment on the stock market. From one investment several results can follow (profit or loss).

propositions to themselves. $\mathcal{L}$ substitutions can be represented as sets of bindings

$$\{q_1 \mapsto \varphi_1, \ldots, q_n \mapsto \varphi_n\}$$

where all the q_i are different. If σ is a $\mathcal{L}$ substitution, then the set $\{q \in Prop \mid \sigma(q) \neq q\}$ is called its domain, denoted $dom(\sigma)$. Use ϵ for the identity substitution. Let $subst_{\mathcal{L}}$ be the set of all $\mathcal{L}$ substitutions.

Definition 3.10. Given a coloring C, C substitutions are of type $Ag \to Ag$ that map all but a finite number of agents $i \in Ag$ to themselves. C substitutions can be represented as sets of bindings

$$\{i_1 \mapsto j_1, \ldots, i_n \mapsto j_n\}$$

where all the i_k are different. If σ^C is a C substitution, then the set $\{i \in Ag \mid \sigma_C(i) \neq i\}$ is called its domain, denoted $dom(\sigma^C)$. Use ϵ for the identity substitution. Let $subst_C$ be the set of all C substitutions.

Definition 3.11. π substitutions are functions of type $\pi \to \pi$ that map all but a finite number of basic relations to themselves. π substitutions can be represented as sets of bindings

$$\{p_1 \mapsto \pi_1, \ldots, p_n \mapsto \pi_n\}$$

where all the p_i are different. If σ^π is a π substitution, then the set $\{p \in P \mid \sigma^\pi(p) \neq p\}$ is called its domain, denoted $dom(\sigma^\pi)$. Use ϵ for the identity substitution. Let $subst_\pi$ be the set of all π substitutions.

Definition 3.12. If $M = \langle W, P, R, C, V\rangle$ is a model and σ is a $\mathcal{L}$ substitution (for an appropriate epistemic language $\mathcal{L}$), then V_M^σ is the valuation given by $\lambda q.[\![\sigma(q)]\!]^M$. In other words, V_M^σ assigns to q the set of worlds w in which $\sigma(q)$ is true. For $M = \langle W, P, R, C, V\rangle$, call M^σ the model given by $\langle W, P, R, C, V_M^\sigma\rangle$.

Definition 3.13. An update model for a finite set of agents N with a language $\mathcal{L}$ is an septuple $U = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C\rangle$ where

- $E = \{e_0, \ldots, e_n\}$ is a finite non-empty set of events,
- $Q_P : Ag \to \mathcal{P}(E^2)$ assigns an accessibility relation $Q_P(i)$ to each agent $i \in Ag$,
- $Q_R : R \to \mathcal{P}(E^2)$ assigns an accessibility relation $Q_R(r)$ to each relation $r \in R$,
- $pre : E \to \mathcal{L}$ assigns a precondition to each event,
- $sub : E \to subst_{\mathcal{L}}$ assigns a $\mathcal{L}$ substitution to each event,
- $sub_P : E \to subst_\pi$ assigns a π substitution to each event,

- $sub_C : E \to subst_C$ assigns a C substitution to each event.

A distinctive update model is a pair U, e, i.e., an update model with a distinguished actual event e.

Remark 3.14. We have included a substitution on the control function C as well. We will need this substitution to change control of worlds that are created by product update. If there were no such substitution, we would not be able to add actions for different agents to a model by means of product update.

Definition 3.15. Given a static model $M = \langle W, P, R, C, V \rangle$, a world $w \in W$, an update model $U = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C \rangle$ and an event $e \in E$ with $M, w \models pre(e)$, we say that the result of executing U, e in M, w is the model $M \circ U, (w, e) = \langle W', P', R', C', V' \rangle$ where

- $W' = \{(v, f) \mid M, v \models pre(f)\}$
- $P'(i) = \{((w_1, e_1), (w_2, e_2)) \mid$ there is a $sub_P(e_1)(i)$ path from (w_1, e_1) to (w_2, e_2) in $\langle W', P'', R', C', V' \rangle\}$
- $P''(i) = \{((v, f), (u, g)) \mid (v, u) \in P_i, (f, g) \in Q_P(i)\}$
- $R' = \{r' \mid r \in R, r' = \{((v, f), (u, g)) \mid (v, u) \in r, (f, g) \in Q_R(r)\}\}$
- $C'((v, f)) = sub_C(f)(C(v))$
- $V'(q) = \{(v, f) \mid M, v \models sub(f)(q)\}$

Definition 3.16. We can add a modality $[A, e]\varphi$ to our logic, where A is an update model containing the event e, with the following interpretation:

$$M, w \models [A, e]\varphi \text{ iff } M, w \models pre(e) \text{ implies } M \circ A, (w, e) \models \varphi$$

Remark 3.17. We have not treated the distinguished, actual state or event in an explicit manner in the definition of (update) models above. We have just supplied information about distinguished states or events together with (update) models. Our (update) models can, however, be easily be modified as to be able to handle more than one distinguished state or event explicitly.

Using this notion of updates we can describe changes in the (static) situation. We will see applications of this in section 4. Also, in section 6.2 we will show that we can use the axiomatization of the static logic for the logic extended with updates. In order to do this we will use certain reduction axioms.

3.3.1 Constraints

In order to maintain adherence to the constraints we laid on our models we must lay certain constraints on our update models as well. The constraints we laid on our models are the constraints of reflexivity of plausibility, distinction, awareness and nondeterminism. In section 6.1 we proof that the following constraints lead to the desired results on update products.

In the following we define η as the unique function that maps all programs in P to corresponding programs in Q_P where all basic programs of the form $?\varphi$ (where φ is no tautology) have been replaced by a program that corresponds to $?\bot$. Furthermore, θ is defined as η with the exception that all basic programs of the form $?\varphi$ (where φ is no falsity) are replaced by $?\top$. Also, $\sim$ is the unique function that maps a relation to its transitive, symmetric closure. Further, let $\xi_i(x) = (?c_i; x; ?c_i) \cup (?\neg c_i; x; ?\neg c_i)$.

We lay the following constraint on update models to make sure the update product doesn't violate the constraint of reflexivity. $\forall e \in E$, $\forall i \in Ag$, $e \rightarrow^{\eta(y)} e$ and y has the property that, for any model, it is reflexive, where $y = x$ if $sub_P(e)(P_i) = \xi_i(x)$ and $y = sub_P(e)(P_i)$ otherwise.

To ensure the update product adheres to the constraint of distinction, we restrict our update models as follows: $\forall e, e' \in E$, $\forall i \in Ag$, if $\exists j \in Ag$ such that $sub_C(e)(j) = i$ and $\exists k \in Ag$ such that $sub_C(e')(k) \neq i$, then $e \not\rightarrow^{\theta(sub_P(e)(P_i))} e'$ or $sub_P(e)(P_i) = \xi_i(x)$ where x is a π-program.

In order to make sure that our update products adhere to the constraint of awareness, we pose the following restriction on update models: $\forall e, e' \in E$, $\forall a \in Q_R$, $\forall i \in Ag$, if $e \rightarrow^{\sim(\theta(sub_P(e)(P_i)))} e'$, $\exists j \in Ag$ such that $sub_C(e)(j) = i$, $\exists k \in Ag$ such that $sub_C(e')(k) = i$ and if $\exists e'' \in E$ such that $e \rightarrow^a e''$, then there is some $E' \subseteq E$ such that $\forall e''' \in E'$ holds $e' \rightarrow^a e'''$ and $\bigvee_{e''' \in E'} pre(e''') \leftrightarrow \top$, and $\forall j \in Ag$, if $sub_C(e)(j) = i$ then $j = i$ and $\forall k \in Ag$, if $sub_C(e')(k) = i$ then $k = i$.

Also, we ensure that our update products do not conflict with the constraint of nondeterminism. In order to do this, we lay the following constraint on our update models. $\forall e, e', e'' \in E$, $\forall r \in Q_R$, $\forall i \in Ag$, if $\exists j \in Ag$ such that $sub_C(e)(j) = i$, if $e \rightarrow^r e'$ and $e \rightarrow^r e''$, then $\forall j \in Ag$, $sub_C(e)(j) = i$ implies $j = i$, and both $e' \rightarrow^{\sim(\eta(y))} e''$ and y has the property that, for any model, its transitive, symmetric closure contains the transitive symmetric closure of P_i, where y is x in case $sub_P(e')(P_i) = \xi_i(x)$ and y is $sub_P(e')(P_i)$ otherwise.

These constraints are strong enough to ensure that the update product adheres to all conditions laid on models in section 3.1.1. There are many (variants on) conditions that can be laid on update models that have this property. These constraints are not per se the

best conditions for this job. They are merely straightforward conditions that do the job, and are conditions that fit the models we use in our analysis.

3.4 Distinction

We are using two different types of actions in the logic described above. Firstly, we have agents' actions in the static models, represented by atomic programs in R. Secondly, we have update actions that change one static model into another. This might seem superfluous, but we make a clear distinction between the two.

The distinction we are making is the distinction between object-level and meta-level. The actions represented by atomic programs are actions in the object-level, and the update actions are actions on the meta-level. In our analysis of social software we make a distinction between the agents involved in a certain procedure or situation, and the designer of the procedure of situation. Since we want to study the behavior of the agents, in order to improve the design, the actions represented by atomic programs are actions on the object-level and the update actions are actions on the meta-level.

4 An Analysis of Bankruptcy

In this section we will consider an example of a game situation in social software where considering beliefs is essential in analyzing the situation. We will show that certain outcomes will be reached with certain beliefs, and other outcomes with other beliefs. We consider only one example, but in general, any situation that can be represented as an extensive-form game can be modeled using our models and logic.

4.1 Bankruptcy game

Consider the bankruptcy game from section 2.1. The model of an instance of this game with three players can be seen in figure 5. We can construct this model using updates as follows.

We begin with the model $\mathcal{B} = \langle W, P, R, C, V \rangle$ where:

- $W = \{\epsilon\}$
- $P^i = \{(\epsilon, \epsilon)\}$, for all $i \in Ag$
- $R = \{c_i \mid i \in Ag, c_i = \{(\epsilon, \epsilon)\}\} \cup \{s_i \mid i \in Ag, s_i = \{(\epsilon, \epsilon)\}\}$
- $C(\epsilon) = \star$

- $V(f) = \{\epsilon\}, V(q) = \emptyset$, for all $q \in Prop, q \neq f$

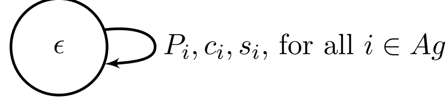


Figure 1: Visual representation of (part of) $\mathcal{B}$.

In order to keep score, we add two counters i^c and i^s, implemented as a large enough finite number of (say n) propositional variables i^c_k and i^s_k, where i^x_k (x ranging over $\{c, s\}$) denotes that at least k agents have performed action x up to this point. An increase of a counter i^x corresponds to the following $\mathcal{L}$ substitution: $\delta^x = \{i^x_0 \mapsto \top, i^x_1 \mapsto i^x_0, \ldots, i^x_n \mapsto i^x_{n-1}\}$. We will use this $\mathcal{L}$ substitution in our update models.

We can add a choice between c_i and s_i for player i using the update model $U(i) = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C\rangle$ where:

- $E = \{p, n, c, s\}$
- $Q_P(i) = \{(p,p),(n,n),(c,c),(s,s),(p,n),(n,p)\}$,
  $Q_P(j) = \{(p,p),(n,n),(c,c),(s,s),(p,n),(n,p),(c,s),(s,c)\}$, for all $j \in Ag, j \neq i$
- $Q_R(c_i) = \{(p,p),(c,c),(s,s),(p,n),(n,c)\}$,
  $Q_R(s_i) = \{(p,p),(c,c),(s,s),(p,n),(n,s)\}$,
  $Q_R(x) = \{(p,p),(c,c),(s,s),(p,n)\}$, for all $x \in R, x \neq c_i, x \neq s_i$
- $pre(p) = \neg f$,
  $pre(n) = pre(c) = pre(s) = f$
- $sub(p) = \{p^c_i \mapsto \bot, p^s_i \mapsto \bot\}$,
  $sub(n) = \{p^c_i \mapsto \bot, p^s_i \mapsto \bot, f \mapsto \bot\}$,
  $sub(c) = \{p^c_i \mapsto \top, p^s_i \mapsto \bot\} \cup \delta^c$,
  $sub(s) = \{p^c_i \mapsto \bot, p^s_i \mapsto \top\} \cup \delta^s$
- $sub_P(p) = \{P_i \mapsto (?c_i; P_i; ?c_i) \cup (?\neg c_i; P_i; ?\neg c_i) \mid i \in Ag\}$,
  $sub_P(e) = \epsilon$, for all $e \in E$, $e \neq p$
- $sub_C(n) = \{i \mapsto \star \mid i \in Ag\} \cup \{\star \mapsto i\}$,
  $sub_C(p) = \epsilon$,
  $sub_C(c) = sub_C(s) = \{j \mapsto \star \mid j \in Ag\}$

Since we kept score, we are able to determine in what terminal worlds the bank went bankrupt (denoted by truth of the atomic proposition b) and in what terminals worlds it did not. We say that the bank will go bankrupt if a majority of agents performed action s instead of action c. This corresponds to saying that the counter i^s is higher than i^c. This

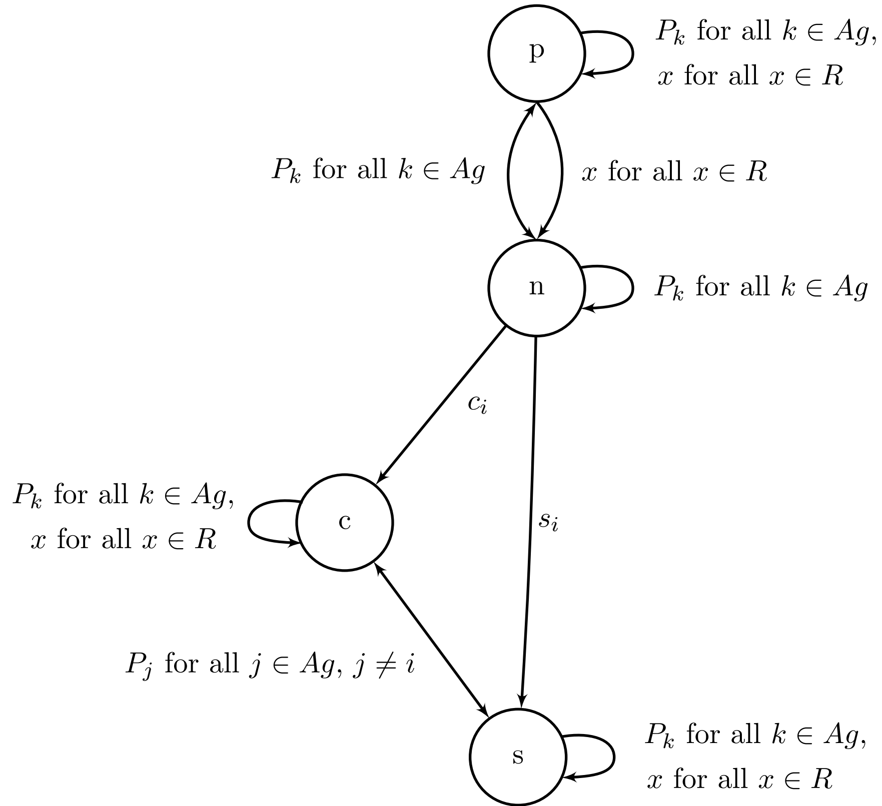


Figure 2: Visual representation of (part of) the first update model.

is true iff $\bigvee_{0<k\leq n} (i^s_k \wedge \neg i^c_k)$ is true. Using this equivalence we add this information to a proposition b in terminal worlds directly.

We can complete our model after adding choices by making closing all relations $r \in R$ in the terminal worlds using the update model $F = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C\rangle$ where:

- $E = \{p, t\}$
- $Q_P(i) = \{(p, p), (t, t), (p, t), (t, p)\}$, for all $i \in Ag$
- $Q_R(c_i) = Q_R(s_i) = \{(p, p), (p, t)\}$, for all $i \in Ag$
- $pre(p) = \neg f$,
  $pre(t) = f$
- $sub(p) = \epsilon$,
  $sub(t) = \{f \mapsto \bot, b \mapsto \bigvee_{0<k\leq n} (i^s_k \wedge \neg i^c_k)\} \cup$
  $\{v^i_n \mapsto p^s_i \vee (p^c_i \wedge \neg b) \mid 1 < n \leq 4, i \in Ag\} \cup$
  $\{v^i_m \mapsto p^c_i \wedge \neg b \mid 4 < m \leq 6, i \in Ag\} \cup$
  $\{v^i_1 \mapsto \top \mid i \in Ag\}$
- $sub_P(p) = \{P_i \mapsto (?c_i; P_i; ?c_i) \cup (?\neg c_i; P_i; ?\neg c_i) \mid i \in Ag\}$
  $sub_P(t) = \epsilon$
- $sub_C(p) = \epsilon$
  $sub_C(t) = \{i \mapsto \star \mid i \in Ag\}$

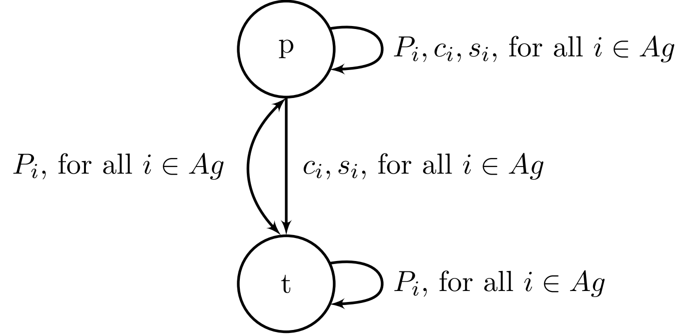


Figure 3: Visual representation of (part of) the second update model.

We can then obtain a (model isomorphic to the) model corresponding to the game from section 2.1 by applying product update: $\Gamma^b = ((((\mathcal{B}, \epsilon \circ U(1), n) \circ U(2), p) \circ U(3), p) \circ F, p$.

Now we can add information about belief by using the following update model. Let c^{-1} denote s and s^{-1} denote c. We can define the update model $P(i, a, G) = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C \rangle$ where $i \in Ag$, $a \in \{c, s\}$ and $G \subset Ag$:

- $E = \{n, c, s\}$
- $Q_P(g) = \{(n, n), (c, c), (s, s), (n, c), (c, n), (n, s), (s, n), (a^{-1}, a)\}$, for all $g \in G$, $Q_P(i) = \{(n, n), (c, c), (s, s), (n, c), (c, n), (n, s), (s, n)\}$, for all $i \in Ag - G$
- $Q_R(c_i) = \{(n, n), (c, c), (s, s), (n, c)\}$, $Q_R(s_i) = \{(n, n), (c, c), (s, s), (n, s)\}$, $Q_R(c_j) = Q_R(s_j) = \{(n, n), (c, c), (s, s)\}$, for all $j \in Ag, j \neq i$
- $pre(n) = \neg p_i^s \wedge \neg p_i^c$, $pre(c) = p_i^c$, $pre(s) = p_i^s$
- $sub(e) = \epsilon$, for all $e \in E$
- $sub_P(e) = \{P_i \mapsto (?c_i; P_i; ?c_i) \cup (?\neg c_i; P_i; ?\neg c_i) \mid i \in Ag\}$, for all $e \in E$
- $sub_C(e) = \epsilon$, for all $e \in E$

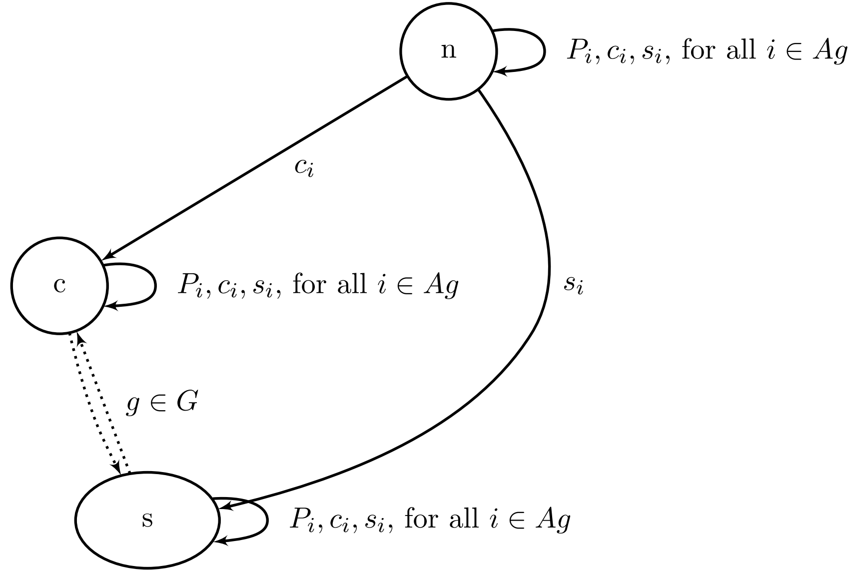


Figure 4: Visual representation of (part of) the third update model.

Updating with the update model $P(i, a, G)$ corresponds to changing beliefs so that it is publicly known that agents G belief that agent i will perform a. We can thus get Γ_S^b by applying the following updates: $(((\Gamma^b, \epsilon \circ P(1, s, \{2, 3\}), n) \circ P(2, s, \{1, 3\}), n) \circ P(3, s, \{1, 2\}), n)$.

Similarly, we can get Γ^b_C by applying the following updates: $(((\Gamma^b, \epsilon \circ P(1, c, \{2, 3\}), n) \circ P(2, c, \{1, 3\}), n) \circ P(3, c, \{1, 2\}), n)$.

4.2 An algorithm

Again, consider the bankruptcy game from section 2.1. The model of an instance of this game with three players can be seen in figure 5. This model corresponds to the extensive form game of the same situation, with payoffs equal to the numbers next to the terminal worlds in figure 5. In this game, there are two Nash equilibria. However, given the beliefs in figure 5, only one equilibrium will be reached. We will show this.

Let Act_w denote the set of possible actions that are to be performed at world $w \in W$. Let $T_{\mathcal{M}}$ denote the set of terminal worlds in $\mathcal{M}$: $\{x \mid x \in W,\ \mathcal{M}, x \models \tau\}$, abbreviated T if it is clear what model $\mathcal{M}$ is discussed. We assign a valuation to every terminal world for every player: $u\colon\ T \times Ag \to \mathbb{N}$.

We can use the following algorithm to assign a minimal expected value to an action a in a world w_0 according to the beliefs of a certain agent i.

- From world w_0 follow $>^{max}_i$ to world w'_0.
- Perform action a in world w_0, leading to world w_1.
- Then, iterate through the following loop until a terminal world is reached.
  1. From world w_n follow $>^{max}_{C_{w_n}}$ to world w'_n.
  2. Perform any optimal action for player $C(w'_n)$ (an action in $opt(w'_n)$) in world w_n, leading to world w_{n+1}.
- When a terminal world w_t is reached, we obtain the value of performing action a in w_0 (denoted $E(w_0, a)$) by obtaining the value of $u(C(w_0), w_t)$.

In certain cases there are multiple possibilities. Sometimes there are more most plausible worlds. In such cases, calculate all possibilities, and take the minimum of all resulting values. Hence the name minimal expected value.

We define $opt(w) = \operatorname*{argmax}_{x \in Act_w} E(w, x)$, where $w \in W - T$. We say an action a is belief-optimal for an agent i in a world w, denoted $\Pi(i, w, a)$, iff $a \in opt(w)$ and $i = C(w)$.

In this algorithm a certain kind of backward induction is used to determine what the optimal action is. For the inductive step the most plausible world is used to determine what action to perform next. However, we use the actual world to determine what the outcome will be. Thus, what actions agents perform is determined by their beliefs and the result of these actions is determined by what is actually the case.

According to these definitions, $\Pi(1, \epsilon, s_1)$, $\Pi(2, s_1, s_2)$ and $\Pi(3, s_1s_2, s_3)$ all hold in the model Γ^b_S in figure 5. This shows that all agents would choose s_i if using (weak) belief to decide what to do.

In the model Γ^b_C in figure 6, where all agents have confidence that all the other agents will choose c_i, $\Pi(1, \epsilon, c_1)$, $\Pi(2, c_1, c_2)$ and $\Pi(3, c_1c_2, c_3)$ all hold. This shows that in this situation all agents would choose c_i if using (weak) belief to decide what to do.

Interestingly, in the model in figure 7, where agent 1 beliefs that agent 2 beliefs all agents have confidence, agent 1 beliefs that agent 3 beliefs all agents are suspicious and agent 1 has no further beliefs, agent 1 will choose the action that will benefit him most. This is c_1, since $\Pi(1, \epsilon, c_1)$ holds. Thus, in cases where an agent beliefs to be crucial in determining the outcome, he will choose the action that according to his beliefs will lead to the outcome most beneficial for him.

4.3 A Logical Approach

The approach in section 4.2 uses a notion of valuations of terminal worlds that goes beyond the logic we defined in section 3. Also, the algorithm is defined on models without using the logic. However, as we will see in this section, for the finite case we can perform the analysis of section 4.2 using formulas. We begin with several auxiliary definitions before performing the analysis.

In order to simulate valuations on terminal worlds we introduce (a bounded number of) extra atomic propositions v^i_n ($i \in Ag$, $0 \leq n < m$ for a certain $m \in \mathbb{N}$). Let N denote $\{n \in \mathbb{N} \mid 0 \leq n < m\}$. We give these propositions the following interpretation:

$$M, t \models v^i_n \text{ iff } M, t \models \tau \text{ and } u(i, t) \leq n$$

Note that τ is the formula that holds in all and only in terminal worlds. We thus dispose of the valuation u on terminal worlds, and place this information in the valuation of these atomic propositions v^i_n.[2] Namely, it is the case that v^i_n holds in a terminal world iff agent i's valuation of that world is at least n. Also let v^i_m denote $\perp$.

Definition 4.1. We let $\bigcup$ denote the union of programs. Given a set $S = \{s_1, \ldots, s_n\}$ and a function f from S to programs, let $\bigcup_{s \in S} f(s)$ denote the program $f(s_1) \cup \ldots \cup f(s_n)$. Also,

[2]Our axiomatization can be easily extended in order to remain complete for this addition to our models. What is needed for that are the following two axiom schemes:

1. $v^i_n \to \tau$, for all $i \in Ag$ and all $n \in \mathbb{N}$ s.t. $0 \leq n < m$.
2. $v^i_{n+1} \to v^i_n$ for all $i \in Ag$ and all $n \in \mathbb{N}$ s.t. $0 \leq n < m - 1$.

We will leave this modification of the completeness proof as an exercise to the reader.

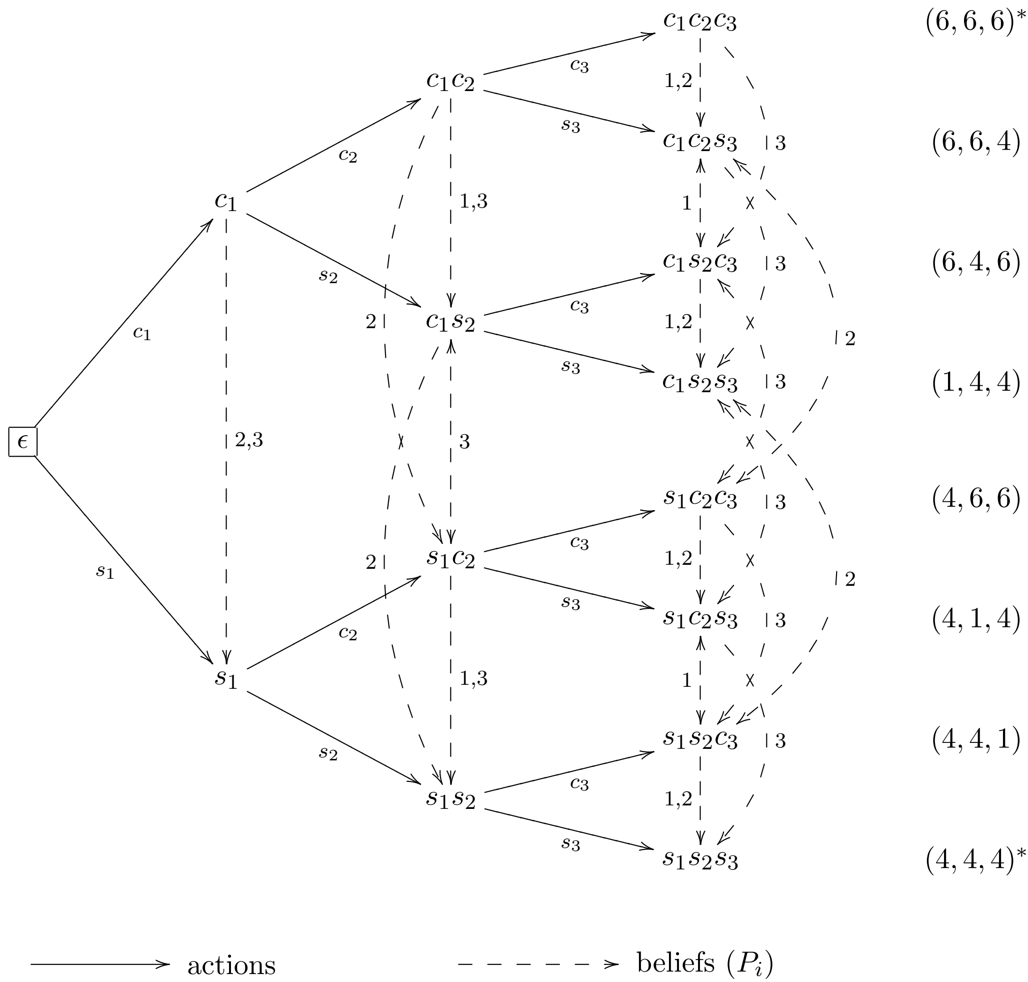


Figure 5: Model for the bankruptcy game with suspicion (Γ_S^b), including preference values on terminal worlds. Note that the reflexive plausibility relations, which are present in all worlds for all agents, are left out in the figure. The worlds that make b true (denoting 'the bank went bankrupt') are $c_1s_2s_3$, $s_1c_2s_3$, $s_1s_2c_3$ and $s_1s_2s_3$.

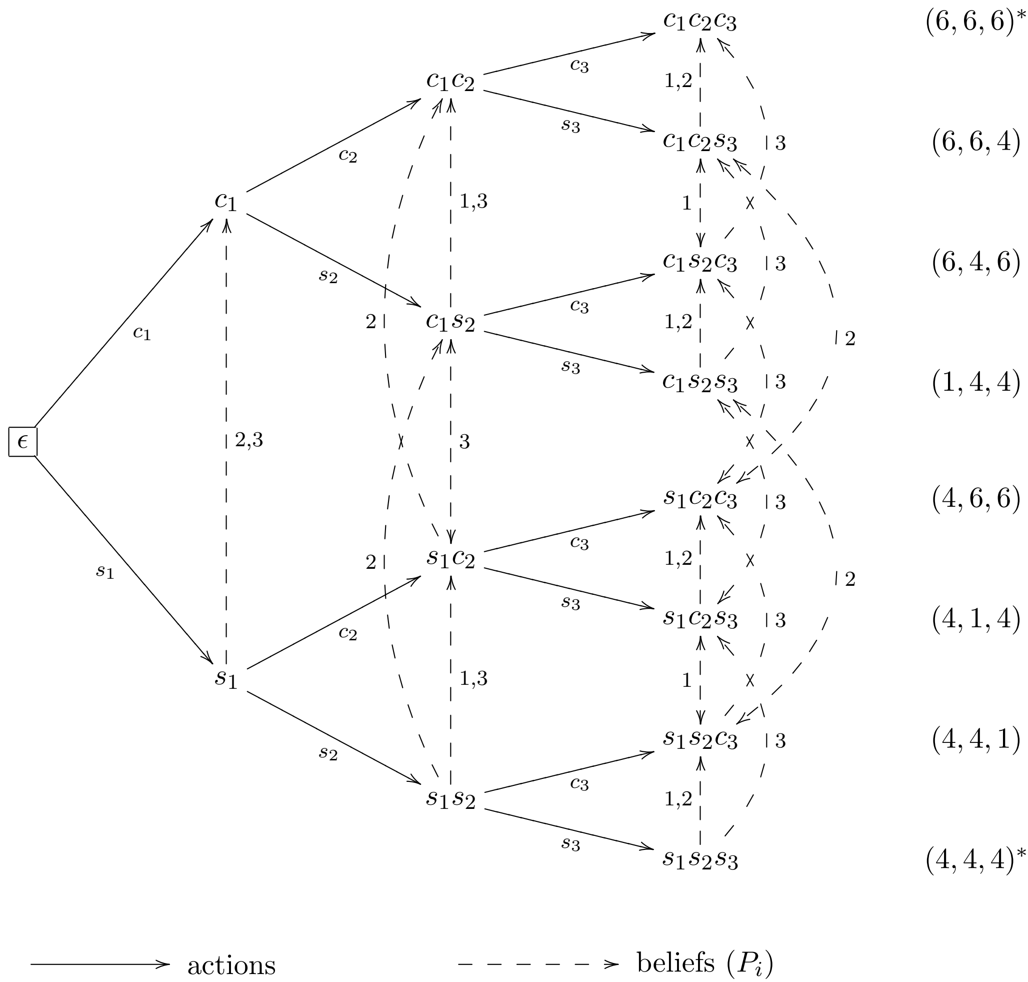


Figure 6: Model for the bankruptcy game with confidence (Γ_C^b), including preference values on terminal worlds. Note that the reflexive plausibility relations, which are present in all worlds for all agents, are left out in the figure. The worlds that make b true (denoting 'the bank went bankrupt') are $c_1s_2s_3$, $s_1c_2s_3$, $s_1s_2c_3$ and $s_1s_2s_3$.

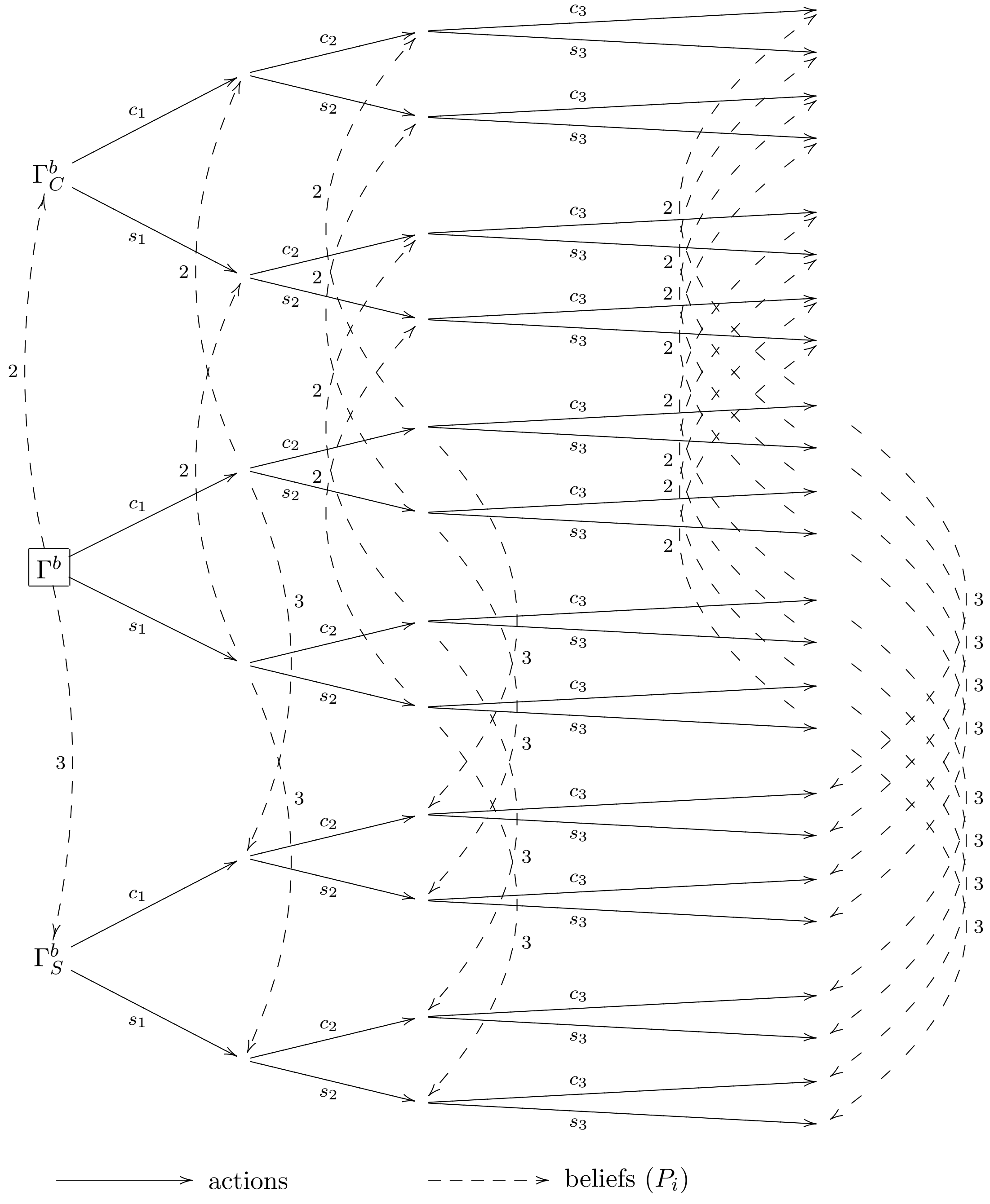


Figure 7: Model for the bankruptcy game where player 2 has confidence, player 3 is suspicious and player 1 knows this and has no further beliefs. Note that the reflexive relations, which are present in all worlds for all agents, are left out in the figure. Here the upper (sub)game (Γ^b_C) is as in figure 6, the lower (sub)game (Γ^b_S) is as in figure 5 and the middle (sub)game (Γ^b) is the game with the union of relations from games Γ^b_C and Γ^b_S.

given an equation $g(s)$, let $\bigcup_{g(s)} f(s)$ denote the program $\bigcup_{s \in S'} f(s)$ where $S' = \{s \mid g(s)\}$.

Definition 4.2. We let $\sum$ denote the concatenation of programs. $\sum_{s \in S} f(s)$ is defined analogously to $\bigcup_{s \in S} f(s)$ and $\sum_{g(s)} f(s)$ is defined analogously to $\bigcup_{g(s)} f(s)$.

We can now define the following recursive pseudo-α-program representing one step in our algorithm. This is the inductive step that represents a belief-optimal move for the player that controls a certain world.

$$\begin{aligned} opt = (?\tau) \cup (?\neg\tau; \bigcup_{i \in Ag} \bigcup_{h \in N} \bigcup_{a_x \in R} (?c_i; ?[>^i_{max}][a^i_x; opt]v^i_h; \\ \sum_{a_y \in R, a_y \neq a_x} ?\neg\langle>^i_{max}\rangle\langle a^i_y; opt\rangle v^i_{h+1}; a^i_x; opt)) \end{aligned} \quad (1)$$

In a terminal world the program is a plain reflexive relation. In a non-terminal world the program executes the optimal action. It does this as follows. In the program the union is taken over all agents $i \in Ag$, over all values $h \in N$ and all actions $a_x \in R$. For the agent that controls the current world (determined by $?c_i$) the program determines whether from the world that is most plausible for the agent that controls the current world there is an action for which the optimum (that is determined recursively) results in a certain value v^i_h such that there is no other action for which the optimum results in a higher value (v^i_{h+1}). This action is then executed, and the optimum is determined for the world that is reached by performing this action.

This recursive pseudo-α-program can be unfolded to a complete α-program for the finite case (finite cases are those in which Ag, R and N are finite). Let μ^m denote the unfolding of this pseudo-program for which the depth is bound to m and for which the innermost occurances of *opt* are replaced by $?\tau$. We can obtain a complete program μ^k that has the right interpretation for our purposes by choosing a large enough $k \in \mathbb{N}$. Let μ denote such a μ^k.

We can then define a formula $opt(i, a)$ (interpreted as a is the optimal move for agent i in world w) as follows.

$$opt(i, a) = \neg\tau \wedge c_i \wedge \langle a\rangle\top \wedge \bigvee_{h \in N} ([>^i_{max}][a; \mu]v^i_h \wedge \bigwedge_{b \in R, b \neq a} \neg\langle>^i_{max}\rangle\langle b; \mu\rangle v^i_{h+1}) \quad (2)$$

This formula is true in all non-terminal worlds controlled by i in which performing a results (when all agents perform their belief optimal actions) in a certain value h such that performing any other action $b \neq a$ does not result in a higher value.

Using this approach, we see that in model Γ^b_S (figure 5) the following holds: $\epsilon \models opt(1, s_1)$, $s_1 \models opt(2, s_2)$ and $s_1 s_2 \models opt(3, s_3)$. Thus, all players would choose s_i if using (weak) belief to decide what to do. Conversely, in model Γ^b_C (figure 6) the following holds: $\epsilon \models opt(1, c_1)$, $c_1 \models opt(2, c_2)$ and $c_1 c_2 \models opt(3, c_3)$. In this case all players would choose c_i if using (weak) belief to decide what to do. Again, in the model in figure 7 agent 1 will perform c_1: $\epsilon \models opt(1, c_1)$.

Also, we can see that in model Γ^b_S, for instance, holds $\epsilon \models [>^1_{max}][(s_1 \cup c_1); \mu]b$, denoting that in world ϵ player 1 believes that (if all players act optimally) whatever action he chooses, the bank will go bankrupt. Conversely, in model Γ^b_C holds $\epsilon \models [>^1_{max}][(s_1 \cup c_1); \mu]\neg b$.

4.4 Turning the tide

The previous analyses show that in a certain situation all agents believe that the bank will go bankrupt, regardless of their individual actions, and in another situation all agents believe that the bank will not go bankrupt, regardless of their actions. In the former case the bank will go bankrupt because of these beliefs (that is, if all agents act according to their beliefs trying to optimize their payoff), and in the second case the bank will not go bankrupt. The former situation is highly undesirable, and the latter is quite desirable. We will show that there are certain update actions (that can be performed by the meta-player, or higher authority, i.e., the government or monetary authority) that can turn the former situation into the latter and vice versa.

4.4.1 Reversing belief

Consider the model Γ^b_S in figure 5. Then consider the following update model $U_1 = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C \rangle$ where

- $E = \{e\}$
- $Q_P = \{(e, e)\}$
- $Q_R = \{(e, e)\}$
- $pre(e) = \top$
- $sub(e) = \epsilon$
- $sub_P(e)(i) = \xi_i(P_i^{\smile})$, for all $i \in Ag$
  $= (?c_i; P_i^{\smile}; ?c_i) \cup (?\neg c_i; P_i^{\smile}; ?\neg c_i)$, for all $i \in Ag$
- $sub_C(e) = \epsilon$

Updating Γ_S^b, ϵ with U_1, e results in a model $\Gamma_S^b \circ U_1, (\epsilon, e)$ that is isomorphic to model Γ_C^b, ϵ in figure 6. The result of this update with U_1, e is that all preference relations are reversed. Whatever any agent believed to be more plausible in the situation before the update, this agent believes to be less plausible after the update.

Interestingly, if we update the model $\Gamma_S^b \circ U_1, (\epsilon, e)$ with U_1, e, we get a model $(\Gamma_S^b \circ U_1) \circ U_1, ((\epsilon, e), e)$ that is isomorphic with Γ_S^b, ϵ. Reversing plausibility relations twice results in the same situation as not reversing the relations at all. This is what we would expect.

4.4.2 Money guarantee

Reversing plausibility relations is not the only way to prevent agents from performing actions that will make the bank go bankrupt. If in all cases it is in the interest of any agent to perform the action that will contribute to the bank's not going bankrupt, the bank will not go bankrupt. This can be achieved by the following update with model $U_2 = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C \rangle$ where

- $E = \{e\}$
- $Q_P = \{(e, e)\}$
- $Q_R = \{(e, e)\}$
- $pre(e) = \top$
- $sub(e) = \{v_n^i \mapsto (p_i^c \wedge b) \vee v_n^i \mid 1 \leq n \leq 5\}$
- $sub_P(e) = \epsilon$
- $sub_C(e) = \epsilon$

This update corresponds to a guarantee of money that governments often offer in cases where banks might go bankrupt. Even if an agent keeps his money in the bank and the bank goes bankrupt, the agent does not lose his money (up to a certain amount). We assume that every agent prefers the bank not going bankrupt over the bank going bankrupt, and that every agent prefers keeping his money in the bank and being guaranteed his money by the government over taking the money to his own private piggy bank. Under these assumptions U_2, e corresponds to such a guarantee.

We can see that such a guarantee is enough to prevent agents from taking their money out of the bank and to keep the bank out of bankruptcy. In the model $\Gamma_S^b \circ U_2, (\epsilon, e)$ the following properties hold: $(\epsilon, e) \models opt(1, c_1)$, $(c_1, e) \models opt(2, c_2)$, $(c_1c_2, e) \models opt(3, c_3)$ and $(c_1c_2c_3, e) \models \neg b$.

4.4.3 Real world correspondence

Using update models U_1 and U_2 it is easy to see that certain properties hold, and to verify that in situations that have been updated with U_1 or U_2 agents will prefer certain actions over other actions. This might seem as if we can easily apply with certain updates to give us the desired results in the real world. However, it is extremely difficult to find a real world action that a government (or any other meta player) can perform that corresponds to certain update models. For instance, what update model U_1 does is reversing all plausibility relations. No government, however, has a way of reversing the belief of all agents involved. These update models thus represent highly idealized changes in reality.

4.5 Advantages and disadvantages

Our analysis of this situation has several advantages and several disadvantages. We will discuss some of these here.

One advantage of the use of our logic in the analysis of this situation is that we have explicit control over agents' beliefs. These beliefs, unlike in many game-theoretical models, do not have to be rational. We laid only several basic constraints on these beliefs (in section 3.1.1). This is an advantage because we can model certain irrational beliefs as well. It is very doubtful that people's beliefs are always rational. Therefore, it is important in analyses of social software to be able to model irrational beliefs as well.

A disadvantage of the use of our logic in the analysis of this example (or in general in the analysis of situations in which actions are effectively performed simultaneously) is that there is a certain asymmetry in the model with respect to the situation that the agents are in. In the situation the agents are all faced with a similar choice, i.e. to choose between two options. We model this as a sequence of decisions in which the agents have no knowledge of what the other agents will do or have done. This results in a plausible model in which we have a means of talking about agents' beliefs of other agents actions. However, we base the analysis of an agent's decision on the beliefs of the agents that are to choose after this agent, but we do not base this analysis on the beliefs of the agents that chose before this agent. All information about beliefs of the agents that chose before this agent has to be encoded in the belief of this agent.

We can ensure that the encoding of this information in the belief of this agent is consistent with the analysis based on the beliefs of agents that are to choose, by laying the following constraint on beliefs in our models. If two sequences of actions differ only with respect to one action (i.e., the first sequence contains an action a where the second sequence contains an action b) and if in world w, controlled by agent i, where the choice between these two action is to be made, holds $opt(i, a)$, then all agents j that cannot distinguish outcomes

s (from the sequence containing a) and s' (from the sequence containing b) must strictly prefer state s over s'.

It is thus not impossible to work around this asymmetry in our modeling, but it is certainly disadvantageous that this should be done in such a fashion.

5 Soundness and Completeness of the Static Logic

Proving soundness of our axioms is pretty straightforward. It can easily be seen that all axioms are validities in every model and that all rules hold in all models as well.

Our completeness proof is based on the completeness proof of PDL without converse in [KP] and [BdRV] and the completeness proof of PDL with converse in [P_2]. We will prove completeness ($\models \varphi$ implies $\vdash \varphi$) by contraposition. Suppose $\nvdash \varphi$. Then $\neg\varphi$ is consistent. We will show that every consistent formula has a model $\mathcal{M}_C$ that makes it true in some world w. Thus $\mathcal{M}_C, w \models \neg\varphi$. Then $\not\models \varphi$. This is a standard Henkin construction for completeness proofs.

We will use several lemmae to prove that every consistent formula has a model that makes it true in some world.

We use a slightly modified version of the Fischer-Ladner closure of a PDL formula ψ that was introduced in [FL], which we denote by $cl(\psi)$. We define $cl(\psi)$ to be the smallest set of formulas containing ψ and containing c_i for all $i \in Ag \cup \{\star\}$ such that:

1. If $\varphi_1 \vee \varphi_2 \in cl(\psi)$, then $\varphi_1 \in cl(\psi)$ and $\varphi_2 \in cl(\psi)$.
2. If $\neg\varphi \in cl(\psi)$, then $\varphi \in cl(\psi)$.
3. If $\langle\pi\rangle\varphi \in cl(\psi)$, then $\varphi \in cl(\psi)$.
4. If $\langle\pi_1 \cup \pi_2\rangle\varphi \in cl(\psi)$, then $\langle\pi_1\rangle\varphi \in cl(\psi)$ and $\langle\pi_2\rangle\varphi \in cl(\psi)$.
5. If $\langle\pi_1;\pi_2\rangle\varphi \in cl(\psi)$, then $\langle\pi_1\rangle\langle\pi_2\rangle\varphi \in cl(\psi)$.
6. If $\langle\pi^*\rangle\varphi \in cl(\psi)$, then $\langle\pi\rangle\varphi \in cl(\psi)$ and $\langle\pi\rangle\langle\pi^*\rangle\varphi \in cl(\psi)$.
7. If $\langle ?\chi\rangle\varphi \in cl(\psi)$, then $\chi \in cl(\psi)$.
8. Conditions 3 through 7 also hold when programs α are substituted for π.

We will use this closure to obtain finite maximal consistent sets. This is necessary because of the non-compactness of PDL.[3]

[3]The following infinite set Γ illustrates this property of non-compactness. Every finite subset $\Gamma' \subseteq \Gamma$ is satisfiable, but Γ is not.

$$\Gamma = \{\neg\varphi, \neg\langle\pi\rangle\varphi, \neg\langle\pi\rangle\langle\pi\rangle\varphi, \ldots\} \cup \{\langle\pi^*\rangle\varphi\}$$

Axioms

1. any tautology of propositional logic
2. $\langle\gamma_1 \cup \gamma_2\rangle\varphi \leftrightarrow \langle\gamma_1\rangle\varphi \vee \langle\gamma_2\rangle\varphi$
3. $\langle\gamma_1; \gamma_2\rangle\varphi \leftrightarrow \langle\gamma_1\rangle\langle\gamma_2\rangle\varphi$
4. $\varphi \vee \langle\gamma\rangle\langle\gamma^*\rangle\varphi \rightarrow \langle\gamma^*\rangle\varphi$
5. $\varphi \rightarrow [\gamma^*](\varphi \rightarrow [\gamma]\varphi) \rightarrow [\gamma^*]\varphi$
6. $\langle\psi?\rangle\varphi \leftrightarrow \psi \wedge \varphi$
7. $\langle(\pi_1 \cup \pi_2)^{\breve{}}\rangle\varphi \leftrightarrow \langle{\pi_1}^{\breve{}} \cup {\pi_2}^{\breve{}}\rangle\varphi$
8. $\langle(\pi_1; \pi_2)^{\breve{}}\rangle\varphi \leftrightarrow \langle{\pi_2}^{\breve{}}; {\pi_1}^{\breve{}}\rangle\varphi$
9. $\langle(\pi^*)^{\breve{}}\rangle\varphi \leftrightarrow \langle(\pi^{\breve{}})^*\rangle\varphi$
10. $\langle(?\psi)^{\breve{}}\rangle\varphi \leftrightarrow \langle?\psi\rangle\varphi$
11. $[\gamma](\varphi \rightarrow \psi) \rightarrow ([\gamma]\varphi \rightarrow [\gamma]\psi)$
12. $\varphi \rightarrow [p]\langle p^{\breve{}}\rangle\varphi$
13. $\varphi \rightarrow [p^{\breve{}}]\langle p\rangle\varphi$
14. $\varphi \rightarrow \langle p\rangle\varphi$
15. $\bigvee_{i \in Ag \cup \{\star\}}(c_i)$
16. $c_i \rightarrow \neg c_j$, for all $i, j \in Ag \cup \{\star\}$ for which holds $i \neq j$
17. $c_i \rightarrow [\sim_i]c_i$, for all $i \in Ag \cup \{\star\}$
18. $(c_i \wedge \langle a\rangle\top) \rightarrow [\sim_i]\langle a\rangle\top$
19. $\langle P_i^s\rangle\varphi \leftrightarrow (\langle P_i^*\rangle\varphi \wedge \neg\langle(P_i^*)^{\breve{}}\rangle\varphi)$
20. $(c_i \wedge \langle a\rangle\varphi) \rightarrow [a]\langle P_i \cup (P_i)^{\breve{}}\rangle\varphi$

Rules

MP if $\vdash \varphi$ and $\vdash \varphi \rightarrow \psi$ then $\vdash \psi$

GEN if $\vdash \varphi$ then $\vdash [\gamma]\varphi$

All axiom schemes and rules that use γ hold for both relations π and α. Axiom scheme 10 only holds for relations π.

Figure 8: A complete axiomatization for our logic.

Remark 5.1. We can assume without loss of generality that the converse operator is only applied to atomic programs, since axioms 7 through 10 provide us with the following equivalences: $(\pi_1 \cup \pi_2)^{\smile} = \pi_1{}^{\smile} \cup \pi_2{}^{\smile}$, $(\pi_1; \pi_2)^{\smile} = \pi_2{}^{\smile}; \pi_1{}^{\smile}$, $(\pi^*)^{\smile} = (\pi^{\smile})^*$ and $(?\chi)^{\smile} = ?\chi$. Using these equivalences we can rewrite any formula in which the converse-operator is applied to complex programs to a formula in which the converse-operator is only applied to atomic programs.

Definition 5.2. Let $cl(\psi) \subseteq \mathcal{L}$ be the closure of some formula. Γ is maximal consistent in $cl(\psi)$ iff

1. $\Gamma \subseteq cl(\psi)$
2. Γ is consistent: $\Gamma \nvdash \bot$
3. Γ is maximal in $cl(\psi)$: there is no $\Gamma' \subseteq cl(\psi)$ such that $\Gamma \subset \Gamma'$ and $\Gamma' \nvdash \bot$

Definition 5.3. We define an atom $\hat{\Gamma}$ to be the conjunction of all formulas in a maximal consistent set in ψ: $\hat{\Gamma} = \bigwedge \Gamma$.

Lemma 5.4. *For all formulas φ in $cl(\psi)$ and all maximal consistent sets Γ in $cl(\psi)$ holds $\varphi \in \Gamma$ iff $\vdash \hat{\Gamma} \rightarrow \varphi$.*

Proof. Since $\varphi \in cl(\psi)$ we know that $\vdash \hat{\Gamma} \rightarrow \varphi$ holds iff φ is one of the conjuncts in $\hat{\Gamma}$. By definition of $\hat{\Gamma}$ this holds iff $\varphi \in \Gamma$. □

Definition 5.5. We define the canonical model $\mathcal{M}_C = \langle W_C, P_C, R_C, C_C, V_C \rangle$ for a certain formula ψ with closure $cl(\psi)$ as follows:

- $W_C = \{\Gamma \mid \Gamma \text{ is maximal consistent in } cl(\psi)\}$
- for all $i \in Ag$, $\Gamma \rightarrow^{P_C(i)} \Delta$ iff $\hat{\Gamma} \wedge \langle P_C(i) \rangle \hat{\Delta}$ is consistent
- for all $i \in Ag$, $\Gamma \rightarrow^{P_C(i)^{\smile}} \Delta$ iff $\hat{\Gamma} \wedge \langle P_C(i)^{\smile} \rangle \hat{\Delta}$ is consistent
- for all $r \in R_C$, $\Gamma \rightarrow^{r} \Delta$ iff $\hat{\Gamma} \wedge \langle r \rangle \hat{\Delta}$ is consistent
- for all $\psi \in cl(\psi)$, $\Gamma \rightarrow^{?\psi} \Gamma$ iff $\psi \in \Gamma$
- $V_C(p) = \{\Gamma \in W_C \mid p \in \Gamma\}$
- for all $\Gamma \in W_C$ and all $i \in Ag$, $C_C(\Gamma) = i$ iff $c_i \in \Gamma$

Remark 5.6. Note that we consider programs of the form $p^{\smile}$ to be atomic programs in the definition above. We will later show that $\Gamma \rightarrow^{p} \Delta$ iff $\Delta \rightarrow^{p^{\smile}} \Gamma$.

Remark 5.7. Note that any maximal consistent set contains exactly one formula of the form c_i, $i \in Ag \cup \{\star\}$. This is because of axioms 15 and 16. If a maximal consistent set would contain no formulas of the form c_i, it would be inconsistent by axiom 15. If a maximal consistent set would contain more than one formula of the form c_i, it would be inconsistent by axiom 16.

Lemma 5.8. ***Lindenbaum lemma.*** *Let $cl(\psi)$ be the closure of some formula. Every consistent subset of $cl(\psi)$ is a subset of a maximal consistent set in $cl(\psi)$.*

Proof. Let $\Delta \subseteq cl(\psi)$ be a consistent set of formulas. Let $|cl(\psi)| = n$. Let $cl(\psi)_k$ be the k-th formula in an enumeration of $cl(\psi)$. Now consider the following sequence of sets of formulas.

$$\begin{array}{lll} \Gamma_0 & = & \Delta \\ \Gamma_{k+1} & = & \begin{cases} \Gamma_k \cup \{cl(\psi)_{k+1}\} & \text{if } \Gamma_k \cup \{cl(\psi)_{k+1}\} \text{ is consistent} \\ \Gamma_k & \text{otherwise} \end{cases} \end{array}$$

It is easy to see that $\Delta \subseteq \Gamma_n$. We show that Γ_n is consistent and maximal in $cl(\psi)$. To see that Γ_n is consistent, observe by induction on k that every Γ_k is consistent. Thus Γ_n is consistent. To see that Γ_n is maximal in $cl(\psi)$, take an arbitrary formula $cl(\psi)_k \in cl(\psi)$ s.t. $cl(\psi)_k \notin \Gamma_n$. Therefore $\Gamma_{k-1} \cup \{cl(\psi)_k\}$ is inconsistent, and so is $\Gamma_n \cup \{cl(\psi)_k\}$. Since $cl(\psi)_k$ was arbitrary there is no $\Gamma' \subseteq cl(\psi)$ such that $\Gamma_n \subset \Gamma'$ and Γ' is consistent. □

Lemma 5.9. *For any program π and all maximal consistent sets Γ and Δ, if $\hat{\Gamma} \wedge \langle\pi\rangle\hat{\Delta}$ is consistent and for all formulas φ within π holds $\varphi \in cl(\psi)$, then $\Gamma \rightarrow^{\pi} \Delta$.*

Proof. We prove this by induction on the complexity of π. The base case $\pi = p$ is by definition of relations p on $\mathcal{M}_C$. We have another base case (0) $\pi = ?\chi$ and three inductive cases: (1) $\pi = \pi_1 \cup \pi_2$, (2) $\pi = \pi_1; \pi_2$ and (3) $\pi = \pi_1^*$.

0. Assume $\hat{\Gamma} \wedge \langle ?\chi\rangle\hat{\Delta}$ to be consistent. Then by axiom 6 $\hat{\Gamma} \wedge \chi \wedge \hat{\Delta}$ is consistent too. Since Γ and Δ are maximal consistent, $\hat{\Delta} = \hat{\Gamma}$ and thus $\Delta = \Gamma$. By assumption χ must be in $cl(\psi)$, so $\chi \in \Gamma$ and by definition of $\rightarrow^{?\chi}$ we know that $\Gamma \rightarrow^{?\chi} \Gamma$.

1. If $\hat{\Gamma} \wedge \langle\pi_1 \cup \pi_2\rangle\hat{\Delta}$ is consistent, then by axiom 2 $(\hat{\Gamma} \wedge \langle\pi_1\rangle\hat{\Delta}) \vee (\hat{\Gamma} \wedge \langle\pi_2\rangle\hat{\Delta})$ is too. Then either $\hat{\Gamma} \wedge \langle\pi_1\rangle\hat{\Delta}$ or $\hat{\Gamma} \wedge \langle\pi_2\rangle\hat{\Delta}$ is consistent, so by the induction hypothesis either $\Gamma \rightarrow^{\pi_1} \Delta$ or $\Gamma \rightarrow^{\pi_2} \Delta$. Therefore $\Gamma \rightarrow^{\pi_1 \cup \pi_2} \Delta$.

2. If $\hat{\Gamma} \wedge \langle \pi_1; \pi_2 \rangle \hat{\Delta}$ is consistent, then by axiom 3 $\hat{\Gamma} \wedge \langle \pi_1 \rangle \langle \pi_2 \rangle \hat{\Delta}$ is too. By the tautology $\bigvee_{E \in W_C} \hat{E}$ and regular modal reasoning[4] we know that $\bigvee_{E \in W_C} (\hat{\Gamma} \wedge \langle \pi_1 \rangle (\hat{E} \wedge \langle \pi_2 \rangle \hat{\Delta}))$ is consistent as well. Thus, for some $E \in W_C$ the formula $\hat{\Gamma} \wedge \langle \pi_1 \rangle (\hat{E} \wedge \langle \pi_2 \rangle \hat{\Delta}))$ is consistent. Then by regular modal reasoning $\hat{\Gamma} \wedge \langle \pi_1 \rangle \hat{E}$ and $\hat{E} \wedge \langle \pi_2 \rangle \hat{\Delta}$ are consistent. By the induction hypothesis then $\Gamma \rightarrow^{\pi_1} E \rightarrow^{\pi_2} \Delta$ and thus $\Gamma \rightarrow^{\pi_1;\pi_2} \Delta$.

3. Suppose $\hat{\Gamma} \wedge \langle \pi_1^* \rangle \hat{\Delta}$ is consistent. Let Λ be the smallest set of atoms containing $\hat{\Gamma}$ such that if $\hat{\Gamma}' \in \Lambda$ and $\hat{\Gamma}' \wedge \langle \pi_1 \rangle \hat{\Delta}'$ is consistent, then $\hat{\Delta}' \in \Lambda$. By the induction hypothesis, if $\hat{\Gamma}' \wedge \langle \pi_1 \rangle \hat{\Delta}'$ is consistent then $\Gamma' \rightarrow^{\pi_1} \Delta'$, so by the definition of Λ, $\Gamma \rightarrow^{\pi_1^*} \Gamma'$ for all Γ' for which $\hat{\Gamma}' \in \Lambda$. We thus only need to show that $\hat{\Delta} \in \Lambda$. Let $\delta = \bigvee \Lambda$. Then $\delta \wedge \langle \pi_1 \rangle \neg \delta$ is inconsistent because by regular modal reasoning it is equivalent to $\bigvee (\hat{\Gamma}' \wedge \langle \pi_1 \rangle \hat{\Delta}')$ where the join is taken over all $\hat{\Gamma}' \in \Lambda$ and all $\hat{\Delta}' \notin \Lambda$, and each such $\hat{\Gamma}' \wedge \langle \pi_1 \rangle \hat{\Delta}'$ is inconsistent by the construction of Λ. Thus $\vdash \delta \rightarrow [\pi_1]\delta$ and by Gen $\vdash [\pi_1^*](\delta \rightarrow [\pi_1]\delta)$ and thus $\vdash \hat{\Gamma} \rightarrow [\pi_1^*](\delta \rightarrow [\pi_1]\delta)$. Also $\vdash \hat{\Gamma} \rightarrow \delta$, and thus by axiom 5 $\vdash \hat{\Gamma} \rightarrow [\pi_1^*]\delta$, and thus $\hat{\Gamma} \wedge \langle \pi_1^* \rangle \neg \delta$ is inconsistent. Because we assumed $\hat{\Gamma} \wedge \langle \pi_1^* \rangle \hat{\Delta}$ to be consistent, $\hat{\Delta}$ must be in Λ, which was to be shown.

This completes the induction on the complexity of π. □

Lemma 5.10. *For any $\langle \pi \rangle \varphi \in cl(\psi)$ and atom $\hat{\Gamma}$, $\vdash \hat{\Gamma} \rightarrow \langle \pi \rangle \varphi$ iff there exists a Δ such that $\Gamma \rightarrow^{\pi} \Delta$ and $\vdash \hat{\Delta} \rightarrow \varphi$.*

Proof. ($\rightarrow$) By regular modal reasoning, if $\vdash \hat{\Gamma} \rightarrow \langle \pi \rangle \varphi$ then $\hat{\Gamma} \wedge \langle \pi \rangle \hat{\Delta}$ is consistent for some atom $\hat{\Delta}$ for which $\vdash \hat{\Delta} \rightarrow \varphi$. Also, by the Fischer-Ladner rules for all formulas φ within π holds $\varphi \in cl(\psi)$. The result follows from lemma 5.9.

($\leftarrow$) We prove this by induction on the complexity of programs π. Firstly, we consider the base cases: (0) $\pi = p$ and (0′) $\pi = ?\chi$. Then we have the inductive cases (1) $\pi = \pi_1; \pi_2$, (2) $\pi = \pi_1 \cup \pi_2$, and (3) $\pi = \pi_1^*$.

0. Suppose there exists Δ such that $\Gamma \rightarrow^{p} \Delta$ and $\vdash \hat{\Delta} \rightarrow \varphi$. By definition of $\rightarrow^{p}$ then $\hat{\Gamma} \wedge \langle p \rangle \hat{\Delta}$ is consistent, and so is $\hat{\Gamma} \wedge \langle p \rangle \varphi$ and thus $\vdash \hat{\Gamma} \rightarrow \langle p \rangle \varphi$.

0′. Suppose there exists Δ such that $\Gamma \rightarrow^{?\chi} \Delta$ and $\vdash \hat{\Delta} \rightarrow \varphi$. By Fischer-Ladner rule 7 we know $\chi \in cl(\psi)$. By definition of $\rightarrow^{?\chi}$ then $\chi \in \Gamma$ and $\Gamma = \Delta$. Since $\vdash \hat{\Gamma} \rightarrow \varphi$ and by lemma 5.4 $\vdash \hat{\Gamma} \rightarrow \chi$ by propositional reasoning $\vdash \hat{\Gamma} \rightarrow \chi \wedge \varphi$ and by axiom 6 then $\vdash \hat{\Gamma} \rightarrow \langle ?\chi \rangle \varphi$.

1. Assume $\vdash \hat{\Delta} \rightarrow \varphi$. If $\Gamma \rightarrow^{\pi_1 \cup \pi_2} \Delta$, then either $\Gamma \rightarrow^{\pi_1} \Delta$ or $\Gamma \rightarrow^{\pi_2} \Delta$. We assume the former without loss of generality. By Fischer-Ladner rule 4 we know that $\langle \pi_1 \rangle \varphi \in$

[4] With regular modal reasoning we mean reasoning with use of axioms 1 and 11 and rules Gen and MP.

$cl(\psi)$. By the induction hypothesis then $\vdash \hat{\Gamma} \to \langle \pi_1 \rangle \varphi$. Since $\vdash \langle \pi_1 \rangle \varphi \to \langle \pi_1 \cup \pi_2 \rangle \varphi$ by axiom 2 we know that $\vdash \hat{\Gamma} \to \langle \pi_1 \cup \pi_2 \rangle \varphi$.

2. Assume $\vdash \hat{\Delta} \to \varphi$. If $\Gamma \to^{\pi_1;\pi_2} \Delta$ then there exists an E such that $\Gamma \to^{\pi_1} E \to^{\pi_2} \Delta$. By the induction hypothesis $\vdash \hat{E} \to \langle \pi_2 \rangle \varphi$. By Fischer-Ladner rules 3 and 5, $\langle \pi_2 \rangle \varphi \in cl(\psi)$. Again by the induction hypothesis $\vdash \hat{\Gamma} \to \langle \pi_1 \rangle \langle \pi_2 \rangle \varphi$ and by axiom 3 then $\vdash \hat{\Gamma} \to \langle \pi_1; \pi_2 \rangle \varphi$.

3. Assume $\vdash \hat{\Delta} \to \varphi$. if $\Gamma \to^{\pi_1^*} \Delta$ then there must be $\Gamma_0, \ldots, \Gamma_n$ such that $\Gamma = \Gamma_0 \to^{\pi_1} \ldots \to^{\pi_1} \Gamma_n = \Delta$. Since $\vdash \hat{\Gamma}_n \to \varphi$ and $\vdash \varphi \to \langle \pi_1^* \rangle$ by axiom 4 $\vdash \hat{\Gamma}_n \to \langle \pi_1^* \rangle \varphi$. By Fisher-Ladner rule 6 $\langle \pi_1 \rangle \langle \pi_1^* \rangle \varphi \in cl(\psi)$ thus by the induction hypothesis $\vdash \hat{\Gamma}_{n-1} \to \langle \pi_1 \rangle \langle \pi_1^* \rangle \varphi$ and thus $\vdash \hat{\Gamma}_{n-1} \to \langle \pi_1^* \rangle \varphi$ by axiom 4. Continuing in this fashion we get $\vdash \hat{\Gamma}_0 \to \langle \pi_1^* \rangle \varphi$.

This completes the induction on the complexity of programs. □

Lemma 5.11. *For any $\langle \alpha \rangle \varphi \in cl(\psi)$ and atom $\hat{\Gamma}$, $\vdash \hat{\Gamma} \to \langle \alpha \rangle \varphi$ iff there exists a Δ such that $\Gamma \to^{\alpha} \Delta$ and $\vdash \hat{\Delta} \to \varphi$.*

Proof. The proof is analogous to the proof of lemma 5.10, substituting programs α for programs π. Also the relevant instantiations of Fischer-Ladner rule 8 are used instead of rules 3, 4, 5, 6 and 7. □

Lemma 5.12. ***Truth lemma.*** *For all $\varphi \in cl(\psi)$ and all maximal consistent Γ in $cl(\psi)$, the following holds: $\varphi \in \Gamma$ iff $\mathcal{M}_C, \Gamma \models \varphi$.*

Proof. We prove this by induction on the complexity of φ. We consider the base case (0) $\varphi = q$, and the inductive cases (1) $\varphi = \neg\varphi_1$, (2) $\varphi = \varphi_1 \wedge \varphi_2$, (3) $\varphi = \langle \pi \rangle \varphi_1$ and (4) $\varphi = \langle \alpha \rangle \varphi_1$.

0. By definition of $\mathcal{M}_C$ we know $q \in \Gamma$ iff $\Gamma \in V_C(q)$. By truth definition $\Gamma \in V_C(q)$ iff $\mathcal{M}_C, \Gamma \models q$. Thus $q \in \Gamma$ iff $\mathcal{M}_C, \Gamma \models q$.

1. By Fischer-Ladner rule 2 we know that $\varphi_1 \in cl(\psi)$ too. Since Γ is consistent, $\neg\varphi_1 \in \Gamma$ iff $\varphi_1 \notin \Gamma$. By the induction hypothesis, $\varphi_1 \notin \Gamma$ holds iff $\mathcal{M}_C, \Gamma \not\models \varphi_1$. By truth definition, $\mathcal{M}_C, \Gamma \not\models \varphi_1$ holds iff $\mathcal{M}_C, \Gamma \models \neg\varphi_1$. Thus $\neg\varphi_1 \in \Gamma$ iff $\mathcal{M}_C, \Gamma \models \neg\varphi_1$.

2. By Fischer-Ladner rule 1 we know that $\varphi_1 \in cl(\psi)$ and $\varphi_2 \in cl(\psi)$. Since Γ is consistent, $\varphi_1 \vee \varphi_2 \in \Gamma$ iff $\varphi_1 \in \Gamma$ or $\varphi_2 \in \Gamma$. By the induction hypothesis $\varphi_1 \in \Gamma$ iff $\mathcal{M}_C, \Gamma \models \varphi_1$ and $\varphi_2 \in \Gamma$ iff $\mathcal{M}_C, \Gamma \models \varphi_2$. By truth definition we know that $\mathcal{M}_C, \Gamma \models \varphi_1$ or $\mathcal{M}_C, \Gamma \models \varphi_2$ iff $\mathcal{M}_C, \Gamma \models \varphi_1 \vee \varphi_2$. By this chain of equivalences, we know that $\varphi_1 \vee \varphi_2 \in \Gamma$ iff $\mathcal{M}_C, \Gamma \models \varphi_1 \vee \varphi_2$.

3. By lemma 5.10 we know that $\vdash \hat{\Gamma} \to \langle\pi\rangle\psi$ iff there exists a Δ such that $\Gamma \to^{\pi} \Delta$ and $\vdash \hat{\Delta} \to \varphi$. By lemma 5.4 then $\langle\pi\rangle\psi \in \Gamma$ iff there exists a Δ such that $\Gamma \to^{\pi} \Delta$ and $\varphi \in \Delta$. By the induction hypothesis $\varphi \in \Delta$ iff $\mathcal{M}_C, \Delta \models \varphi$. By truth definition we know that $\Gamma \to^{\pi} \Delta$ and $\mathcal{M}_C, \Delta \models \varphi$ iff $\mathcal{M}_C, \Gamma \models \langle\pi\rangle\psi$. By this chain of equivalences, we know that $\langle\pi\rangle\psi \in \Gamma$ iff $\mathcal{M}_C, \Gamma \models \langle\pi\rangle\psi$.

4. The argument is analogous to the argument in (3), using lemma 5.11 instead of 5.10.

This completes the induction on the complexity of formulas. □

Lemma 5.13. *The model $\mathcal{M}_C$ constructed for any given formula φ is an action plausibility model.*

Proof. We need to show that the model $\mathcal{M}_C$ given has all properties of action plausibility models, including the property that it adheres to the constraints that we layed on models.

As mentioned before in remarks 5.1 and 5.6, we assumed that the converse operator is only applied to atomic programs and we considered programs of the form $p^{\smile}$ to be atomic programs in their own right. We will show that all programs containing the converse operator in $\mathcal{M}_C$ behave as they should in an action plausibility model. By axioms 7 through 10 all formulas containing the converse operator are provably equivalent to formulas containing only converse operators applied to atomic programs. We show that for all atomic programs p in $\mathcal{M}_C$, $p^{\smile}$ is exactly its converse in $\mathcal{M}_C$ (i.e., $\Gamma \to^{p} \Delta$ iff $\Delta \to^{p^{\smile}} \Gamma$ for all Γ and Δ in W_C). Suppose this weren't the case. Then there exist $\Gamma, \Delta \in W_C$ such that either $\Gamma \to^{p} \Delta$ and $\Delta \not\to^{p^{\smile}} \Gamma$ or $\Gamma \not\to^{p} \Delta$ and $\Delta \to^{p^{\smile}} \Gamma$. In the former case holds $\mathcal{M}_C, \Gamma \models \hat{\Gamma}$ and thus by axiom 12 $\mathcal{M}_C, \Gamma \models [p]\langle p^{\smile}\rangle\hat{\Gamma}$. Then $\mathcal{M}_C, \Delta \models \langle p^{\smile}\rangle\hat{\Gamma}$. We know that $\hat{\Gamma}$ holds only in Γ, but $\Gamma \not\to^{p^{\smile}} \Delta$. Therefore no world Γ' that makes $\hat{\Gamma}$ true is accessible from Δ. This leads to a contradiction. The latter case is completely analogous, with the use of axiom 13 instead of axiom 12. For all atomic programs p, $p^{\smile}$ is thus the exact converse. The converse operator in $\mathcal{M}$ therefore has the correct properties.

We have already seen in remark 5.7 that every maximal consistent set contains exactly one formula of the form c_i. Thus, all worlds in $\mathcal{M}_C$ make exactly one formula of the form c_i true.

We now show that $\mathcal{M}_C$ adheres to the constraint of reflexivity of plausibility. We do this by contradiction. Assume there exists a $w \in W$ and an $i \in Ag$ for which holds $w \not\to^{P_i} w$. Let φ be a formula that is uniquely true in w. Such a formula does exist. (We show this by contradiction. Assume such a formula does not exist. Then there must be another world w' that makes exactly the same formulas true as w. Then w and w' must be the same maximal consistent set, and thus w and w' must be the same world. Therefore, there must be a formula that is only true in w.) Thus $\mathcal{M}_C, w \models \varphi$ and $\mathcal{M}_C, w \models \neg\langle P_i\rangle\varphi$. But then

by axiom 14 also $\mathcal{M}_C, w \models \langle P_i \rangle \varphi$. This is a contradiction, so $\mathcal{M}_C$ must adhere to the constraint of reflexivity of plausibility.

We have to show that $\mathcal{M}_C$ adheres to the constraint that agents can distinguish worlds that they own from worlds that they do not own. We do this by contradiction. Assume an agent cannot make this distinction. Formally, then there must exist worlds $w, w' \in W_C$ and agent $i \in Ag$ such that $w \rightarrow^{\sim_i} w'$ and $\mathcal{M}_C, w \models c_i$ and $\mathcal{M}_C, w' \not\models c_i$. We know then that $\mathcal{M}_C, w' \models \neg c_i$ and by the truth lemma then $\neg c_i \in w'$. By lemma 5.10 then we know that $\langle \sim_i \rangle \neg c_i \in w$. By the truth lemma then $\mathcal{M}_C, w \models \langle \sim_i \rangle \neg c_i$ and thus $\mathcal{M}_C, w \models \neg [\sim_i] c_i$. Then by axioms 1 and 17 and rule MP we know that $\mathcal{M}_C, w \models \bot$, which is a contradiction since w is a maximal consistent set.

Then we need to show that in the model $\mathcal{M}_C$ the correct relation holds between P_i and P_i^s. We will do this by contradiction. We have that $P_i^s = \{\langle x, y \rangle \mid x \rightarrow^{P_i^*} y \wedge \neg(y \rightarrow^{P_i^*} x)\}$. Suppose this equality does not hold. This means that there are $x, y \in W_C$ s.t. (1) $x \rightarrow^{P_i^s} y$ and not $x \rightarrow^{P_i^*} y \wedge \neg(y \rightarrow^{P_i^*} x)$ or (2) $x \rightarrow^{P_i^*} y \wedge \neg(y \rightarrow^{P_i^*} x)$ and not $x \rightarrow^{P_i^s} y$. Let φ be the conjunction of all formulas in y; this formula φ holds only in y. Because we used the Fischer-Ladner closure to determine the maximal consistent sets, φ is a finite formula.

1. In this case $x \rightarrow^{P_i^s} y$ and either $x \not\rightarrow^{P_i^*} y$ or $x \rightarrow^{(P_i^*)^\smile} y$. Then $\mathcal{M}_C, x \models \langle P_i^s \rangle \varphi$ and either $\mathcal{M}_C, x \not\models \langle P_i^* \rangle \varphi$ or $\mathcal{M}_C, x \models \langle (P_i^*)^\smile \rangle \varphi$. In both cases, this leads directly to a contradiction with axiom 19.

2. In this case $x \not\rightarrow^{P_i^s} y$ and both $x \rightarrow^{P_i^*} y$ and $x \not\rightarrow^{(P_i^*)^\smile} y$. Then $\mathcal{M}_C, x \models \langle P_i^s \rangle \varphi$, $\mathcal{M}_C, x \models \langle P_i^* \rangle \varphi$ and $\mathcal{M}_C, x \not\models \langle (P_i^*)^\smile \rangle \varphi$. This leads directly to a contradiction with axiom 19.

Thus in $\mathcal{M}_C$ the correct relation holds between P_i and P_i^s.

Now we have to show that model $\mathcal{M}_C$ adheres to the constraint of awareness. We will prove that the model $\mathcal{M}_C$ has the property of awareness by contradiction. Suppose $\mathcal{M}_C$ does not have the property of awareness. Then there must exist worlds w, w' and w'' in W_C, there must be an $i \in Ag$ and an $a \in R$ such that $\mathcal{M}_C, w \models c_i$, $\mathcal{M}_C, w' \models c_i$, $w \rightarrow^{\sim_i} w'$, $w \rightarrow^a w''$, and there exists no w''' such that $w' \rightarrow^a w'''$. We know that $\top \in w''$, and thus by lemma 5.11 $\langle a \rangle \top \in w$. By the truth lemma we know that $c_i \in w$. Then by axioms 1 and 18 and rule MP we know that $[\sim_i]\langle a \rangle \top \in w$. Now suppose $\langle a \rangle \top \notin w'$, then $\neg \langle a \rangle \top \in w'$. Then by lemma 5.10 $\langle \sim_i \rangle \neg \langle a \rangle \top \in w$. This is a contradiction with the fact that w is consistent and $[\sim_i]\langle a \rangle \top \in w$. Thus $\langle a \rangle \top \in w'$. Then by lemma 5.11 we know that there exists a w''' such that $w' \rightarrow^a w'''$. This is a contradiction with our assumption that the model doesn't have the property of awareness. $\mathcal{M}_C$ thus has the property of awareness.

Then we show that model $\mathcal{M}_C$ adheres to the constraint of determinism. Again, we do this by contradiction. Suppose $\mathcal{M}_C$ violates the constraint of determinism. That means that $\exists w, w', w'' \in W$, $\exists r \in R$, $\exists i \in Ag$, such that $w \rightarrow^r w'$, $w \rightarrow^r w''$, $C(w) = i$ and

neither $w' \to^{P_i} w''$ nor $w'' \to^{P_i} w'$. Let φ be the formula that is true only in w'. Then $\mathcal{M}_C, w \models c_i \wedge \langle r \rangle \varphi$. By axiom 20 then also $\mathcal{M}_C, w \models [r]\langle P_i \cup P_i{}^{\smile} \rangle \varphi$. Thus $\mathcal{M}_C, w'' \models \langle P_i \cup P_i{}^{\smile} \rangle \varphi$. This can only be true if $w'' \to^{P_i \cup P_i{}^{\smile}} w'$, since φ is only true in w'. However, since neither $w' \to^{P_i} w''$ nor $w'' \to^{P_i} w'$, $w'' \not\to^{P_i \cup P_i{}^{\smile}} w'$. This is a direct contradiction. Thus $\mathcal{M}_C$ adheres to the constraint of determinism.

We have thus shown that $\mathcal{M}_C$ is an action plausibility model. □

Theorem 5.14. ***Completeness.*** *The axiomatization given for our logic (figure 8) is semantically complete ($\models \varphi$ implies $\vdash \varphi$).*

Proof. Proving semantic completeness ($\models \varphi$ implies $\vdash \varphi$) can be done by contraposition. Suppose $\not\vdash \varphi$. Then $\neg\varphi$ is consistent. We can construct a model $\mathcal{M}_C$ for $\neg\varphi$ with the correct properties. By lemma 5.8 and lemma 5.12 we know that in the model $\mathcal{M}_C = \langle W_C, P_C, R_C, C_C, V_C \rangle$ constructed for a formula $\neg\varphi$ there must be at least one world w in W_C for which holds $\mathcal{M}_C, w \models \neg\varphi$. Therefore $\not\models \varphi$. □

6 Well-behavedness of the Updates

We show that our updates are well-behaved by proving certain properties of the update models. Firstly, we proof that an update product adheres to the constraints we laid on models. Secondly, we show that the update modality can be reduced to the static logic. Thirdly, we show that our (update) models are well-behaved under bisimulation.

6.1 Preservation of Properties

In order to show that our updates are well-behaved, we must show that the result of updating a model with any update adheres to the constraints we laid on models. We laid four constraints on our models and we laid four corresponding constraints on our update models. In the following proofs we let η, θ, $\sim$ and ξ be as defined in section 3.3.1.

Theorem 6.1. *For any model M and any update model U, the update product $M \circ U$ does not violate the constraints we laid on models in section 3.3.1.*

Proof. We show that the update product adheres to the constraint of reflexivity. We know that in the original model $M = \langle W, P, R, C, V \rangle$ holds $\forall w \in W$, $\forall i \in Ag$, $w \to^{P_i} w$. Since in $U = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C \rangle$ holds that $sub_P(e)(P_i)$ is reflexive for any P_i in any model, we know that $\forall w \in W$, $w \to^{sub_P(e)(P_i)} w$. Also, we laid the constraint on U

that $\forall e \in E$, $e \rightarrow^{\eta(sub_P(e)(P_i))} e$ or $e \rightarrow^{\eta(x)} e$ if $sub_P(e)(P_i)) = \xi_i(x)$. In the former case, since $\eta(sub_P(e)(P_i))$ is a subset of $sub_P(e)(P_i)$, by definition of update we know that in $M \circ U = \langle W', P', R', C', V' \rangle$ holds $\forall (w, e) \in W'$, $(w, e) \rightarrow^{P_i} (w, e)$. In the latter case, a similar argument holds, since $C'((w, e)) = C'((w, e))$.

We show that the update product adheres to the constraint of distinction. We know that in the original model $M = \langle W, P, R, C, V \rangle$ holds $\forall w, w' \in W$, if $C(w) \neq C(w')$ then $w \not\rightarrow^{P_{C(w)}} w'$. Assume that in the update product $M \circ U = \langle W', P', R', C', V' \rangle$ holds that if $C'((w, e)) \neq C'((w', e'))$ for arbitrary (w, e) and (w', e') in W'. By definition of update, if this is the case then $\exists j \in Ag$ such that in $U = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C \rangle$ holds $sub_C(e)(j) = C'((w, e))$ and $\exists k \in Ag$ such that $sub_C(e')(k) \neq C'((w, e))$. Then by the constraint we laid on U we know that $e \not\rightarrow^{\theta(sub(e)(P_{C'((w,e))}))} e'$ or $sub_P(e)(P_{C'((w,e))}) = \xi_{C'((w,e))}(x)$ for some π-program x. In the former case, since $\theta(sub(e)(P_{C'((w,e))}))$ is a superset of $sub(e)(P_{C(w)})$, by definition of product update holds $(w, e) \not\rightarrow^{P_{C'((w,e))}} (w', e')$. In the latter case, since $C'((w, e)) \neq C'((w', e'))$ we know by definition of ξ_i that $(w, e) \not\rightarrow^{P_{C'((w,e))}} (w', e')$.

We show that the update product adheres to the constraint of awareness. We know that in the original model $M = \langle W, P, R, C, V \rangle$ this constraint is not violated. Let $U = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C \rangle$. Assume that in $M \circ U = \langle W', P', R', C', V' \rangle$ for arbitrary $(w, e), (w', e') \in W'$ and arbitrary $a \in R'$ holds that $C'((w, e)) = C'((w', e'))$, that $(w, e) \rightarrow^{\sim C'((w,e))} (w', e')$ and that $\exists (w'', e'') \in W'$ such that $(w, e) \rightarrow^{a} (w'', e'')$. Then by definition of product update must hold $e \rightarrow^{\sim(\theta(sub_P(e)(P_{C'((w,e))})))} e'$ (for $\theta(sub_P(e)(P_{C'((w,e))}))$ is a superset of $sub_P(e)(P_{C'((w,e))})$, $e \rightarrow^{a} e''$ and $\exists j, k \in Ag$ such that $sub_C(e)(j) = C'((w, e))$, $sub_C(e')(k) = C'((w, e))$. Then by the constraint we laid on U we know that there exists $E' \subseteq E$ such that $\forall e''' \in E'$ holds $e' \rightarrow^{a} e'''$ and that all $w \in W$ make the preconditions of at least one $e''' \in E'$ true. We also know by this constraint on U that there is a unique $j \in Ag$ such that $sub_C(e)(j) = C'((w, e))$, $sub_C(e')(j) = C'((w, e))$ and $j = C'((w, e))$. Then by definition of product update and by the fact that this unique $j = C'((w, e))$ we know that $C(w) = C'((w, e))$, $w \rightarrow^{\sim C'((w,e))} w'$ and $w \rightarrow^{a} w''$. Since M did not violate the constraint of awareness, we know that $\exists w''' \in W$ such that $w' \rightarrow^{a} w'''$. Therefore $\exists (w''', e''') \in W'$ such that $(w', e') \rightarrow^{a} (w''', e''')$.

We show that the update product adheres to the constraint of nondeterminism. We know that in the original model $M = \langle W, P, R, C, V \rangle$ holds $\forall w, w', w'' \in W$, $\forall a \in R$, if $w \rightarrow^{a} w'$ and $w \rightarrow^{a} w''$ then $w' \rightarrow^{\sim C(w)} w''$. Let $U = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C \rangle$. Assume that in $M \circ U = \langle W', P', R', C', V' \rangle$ for arbitrary $(w, e), (w', e'), (w'', e'') \in W'$ and arbitrary $a \in R'$ that $(w, e) \rightarrow^{a} (w', e')$ and $(w, e) \rightarrow^{a} (w'', e'')$. Then by definition of product update we know that in U holds $e \rightarrow^{a} e'$, $e \rightarrow^{a} e''$ and $\exists j \in Ag$ such that $sub_C(e)(j) = C'((w, e))$. By the constraint we laid on U we know then that there is a unique $j \in Ag$ such that $sub_C(e)(j) = C'((w, e))$ and that holds $j = C'((w, e))$, and that either $e' \rightarrow^{\sim(\eta(sub_P(e)(P_{C'((w,e))})))} e''$ or $sub_P(e')(P_{C'((w,e))}) = \xi_{C'((w,e))}(x)$ and $e' \rightarrow^{\sim(\eta(x))} e''$.

Since this j is unique we know that $C'((w,e) = C(w)$ and thus $w' \rightarrow^{\sim_{C'((w,e))}} w''$. In the former case, we know that $\eta(sub_P(e')(P_{C'((w,e))}))$ is a subset of $sub_P(e')(P_{C'((w,e))})$. By the property of $sub_P(e')(P_{C'((w,e))})$ that its transitive, symmetric closure contains the transitive, symmetric closure of $P_{C'((w,e))}$ in M, we know that $w' \rightarrow^{\sim_{sub_P(e')(P_{C'((w,e))})}} w''$. Therefore, $(w',e') \rightarrow^{\sim_{C'((w,e))}} (w'',e'')$. In the latter case, we know that $C(w') = C(w)$ iff $C(w'') = C(w)$, since $w' \rightarrow^{\sim_{C(w)}} w''$ and the constraint of distinction is not violated. Therefore, an argument analogous to the former case can be made (where x is substituted for $sub_P(e')(P_{C'((w,e))})$), and thus $(w',e') \rightarrow^{\sim_{C'((w,e))}} (w'',e'')$.

We have thus shown that an update product adheres to the constraints we laid on our models in section 3. □

6.2 Reduction of Updates to Static Logic

In order to show that the update modality can be reduced to the static logic, we use the approach in [vBvEK] and in [vEW] for our purposes. We use the following definitions for program transformations T_{ij}^U and path transformers K_{ijk}^U for our α-programs taken directly from [vBvEK].

$$
\begin{aligned}
T_{ij}^U(a) &= \begin{cases} ?pre(e_i); a & \text{if } e_i \rightarrow^a e_j \\ ?\bot & \text{otherwise} \end{cases} \\
T_{ij}^U(?\varphi) &= \begin{cases} ?(pre(e_i) \wedge [U,e_i]\varphi) & \text{if } i = j \\ ?\bot & \text{otherwise} \end{cases} \\
T_{ij}^U(\pi_1;\pi_2) &= \bigcup_{k=0}^{n-1}(T_{ik}^U(\pi_1); T_{kj}^U(\pi_2)) \\
T_{ij}^U(\pi_1 \cup \pi_2) &= T_{ij}^U(\pi_1) \cup T_{ij}^U(\pi_2) \\
T_{ij}^U(\pi^*) &= K_{ijn}^U(\pi) \\
K_{ij0}^U(\alpha) &= \begin{cases} ?\top \cup T_{ij}^U(\alpha) & \text{if } i = j \\ T_{ij}^U(\alpha) & \text{otherwise} \end{cases} \\
K_{ij(k+1)}^U(\alpha) &= \begin{cases} (K_{kkk}^U(\alpha))^* & \text{if } i = k = j \\ (K_{kkk}^U(\alpha))^*; K_{kjk}^U(\alpha) & \text{if } i = k \neq j \\ K_{ikk}^U(\alpha); (K_{kkk}^U(\alpha))^* & \text{if } i \neq k = j \\ K_i^U jk(\alpha) \cup (K_{ikk}^U(\alpha); (K_k^U kk(\alpha))^*; K_{kjk}^U(\alpha)) & \text{otherwise } (i \neq k \neq j) \end{cases}
\end{aligned}
$$

Also, we use the following definitions for program transformations $\underline{T}_{ij}^U$ and path transformers $\underline{K}_{ijk}^U$ for our π-programs taken directly from [vEW].

$$\underline{T}^U_{ij}(a) = \begin{cases} ?pre(e_i); sub(e_i)(a) & \text{if } e_i \to^{sub(e_i)(a)} e_j \text{ in } U \\ ?\bot & \text{otherwise} \end{cases}$$
$$\underline{T}^U_{ij}(a^\smile) = \begin{cases} ?pre(e_i); (sub(e_i)(a))^\smile & \text{if } e_i \to^{(sub(e_i)(a))^\smile} e_j \text{ in } U \\ ?\bot & \text{otherwise} \end{cases}$$
$$\underline{T}^U_{ij}(?\varphi) = \begin{cases} ?(pre(e_i) \wedge [U, e_i]\varphi) & \text{if } i = j \\ ?\bot & \text{otherwise} \end{cases}$$
$$\underline{T}^U_{ij}(\pi_1; \pi_2) = \bigcup_{k=0}^{n-1}(\underline{T}^U_{ik}(\pi_1); \underline{T}^U_{kj}(\pi_2))$$
$$\underline{T}^U_{ij}(\pi_1 \cup \pi_2) = \underline{T}^U_{ij}(\pi_1) \cup \underline{T}^U_{ij}(\pi_2)$$
$$\underline{T}^U_{ij}(\pi^*) = K^U_{ijn}(\pi)$$
$$\underline{K}^U_{ij0}(\pi) = \begin{cases} ?\top \cup \underline{T}^U_{ij}(\pi) & \text{if } i = j \\ \underline{T}^U_{ij}(\pi) & \text{otherwise} \end{cases}$$
$$\underline{K}^U_{ij(k+1)}(\pi) = \begin{cases} (\underline{K}^U_{kkk}(\pi))^* & \text{if } i = k = j \\ (\underline{K}^U_{kkk}(\pi))^*; \underline{K}^U_{kjk}(\pi) & \text{if } i = k \neq j \\ \underline{K}^U_{ikk}(\pi); (\underline{K}^U_{kkk}(\pi))^* & \text{if } i \neq k = j \\ \underline{K}^U_i jk(\pi) \cup (\underline{K}^U_{ikk}(\pi); (\underline{K}^U_k kk(\pi))^*; \underline{K}^U_{kjk}(\pi)) & \text{otherwise } (i \neq k \neq j) \end{cases}$$

Theorem 6.2. *For all update models U and all programs α, the following equivalence holds:*

$$(w, v) \in [\![T^U_{ij}(\alpha); ?\mathrm{pre}(e_j)]\!]^M \text{ iff } ((w, e_i), (v, e_j)) \in [\![\alpha]\!]^{M \circ U}$$

Proof. A proof of this theorem is given in [vBvEK], using models that are essentially the same as our models with respect to α-programs. □

Theorem 6.3. *For all update models U and all programs π, the following equivalence holds:*

$$(w, v) \in [\![\underline{T}^U_{ij}(\pi); ?\mathrm{pre}(e_j)]\!]^M \text{ iff } ((w, e_i), (v, e_j)) \in [\![\pi]\!]^{M \circ U}$$

Proof. In [vEW] a modification of the proof for theorem 6.2 in [vBvEK] is given using program transformations $\underline{T}^U_{ij}$ instead of T^U_{ij}. This proof uses models that are the same as our models with respect to π-programs. □

This leads to the following two reduction axioms, where n is the number of events in update model A,

$$[A, e_i][\alpha]\varphi \leftrightarrow \bigwedge_{j=0}^{n-1} [T^A_{ij}(\alpha)][A, e_j]\varphi$$

$$[A, e_i][\pi]\varphi \leftrightarrow \bigwedge_{j=0}^{n-1} [\underline{T}^A_{ij}(\pi)][A, e_j]\varphi$$

that together with the following reduction axioms (where $q^{sub(e)}$ is $sub(e)(q)$ if q is in the domain of $sub(e)$ and is q otherwise)

$$\begin{array}{rcl} [U, e]\top & \leftrightarrow & \top \\ [U, e]q & \leftrightarrow & (pre(e) \rightarrow q^{sub(e)}) \\ [U, e]\neg\varphi & \leftrightarrow & (pre(e) \rightarrow \neg[U, e]\varphi) \\ [U, e](\varphi_1 \wedge \varphi_2) & \leftrightarrow & ([U, e]\varphi_1 \wedge [U, e]\varphi_2) \end{array}$$

and together with all the axioms and inference rules from the proof system in figure 8 and inference rules of necessitation for all update model modalities constitute proof system $\mathcal{U}$.

Theorem 6.4. *Using proof system $\mathcal{U}$ for our logic with update modalities, $\models \varphi$ iff $\vdash \varphi$ holds.*

Proof. The proof system for our logic without update modalities is complete, and every formula in the language with update modalities is provably equivalent to a formula in the logic without update modalities. □

6.3 Bisimulation

In order to determine whether our models and update models are well-behaved, we define a notion of equivalence. This is a relation between (update) models called bisimulation. We let $M \Leftrightarrow N$ denote that models M and N are bisimilar. Equivalently for update models.

Definition 6.5. We say that models $M = \langle W, P, R, C, V\rangle$ and $M' = \langle W', P', R', C', V'\rangle$ are bisimilar ($M \Leftrightarrow M'$) iff there exists a non-empty binary relation $E \subseteq W \times W'$ such that for all $w \in W$ and all $w' \in W'$, if wEw' then the following holds:

- $C(w) = C(w')$
- $w \in V(q)$ iff $w' \in V'(q)$, for all $q \in Prop$
- for all $i \in Ag$ and all $v \in W$, if $w \rightarrow^{P_i} v$ then $\exists v' \in W'$ s.t. $w' \rightarrow^{P'_i} v'$ and vEv' (zig)
- for all $i \in Ag$ and all $v' \in W'$, if $w' \rightarrow^{P'_i} v'$ then $\exists v \in W$ s.t. $w \rightarrow^{P_i} v$ and vEv' (zag)
- above zig and zag conditions in which $P_i^{\smile}$ is substituted for P_i.
- for all $a \in R$ (with corresponding $a' \in R'$) and all $v \in W$, if $w \rightarrow^{a} v$ then $\exists v' \in W'$ s.t. $w' \rightarrow^{a'} v'$ and vEv' (zig)

- for all $a' \in R'$ (with corresponding $a \in R$) and all $v' \in W'$, if $w' \rightarrow^{a'} v'$ then $\exists v \in W$ s.t. $w \rightarrow^{a} v$ and vEv' (zag)

We say that $M, w \leftrightarroweq M', w'$ iff $M \leftrightarroweq M'$, $w \in W$, $w' \in W'$ and wEw'.

Definition 6.6. We say that update models $U = \langle E, Q_P, Q_R, pre, sub, sub_P, sub_C \rangle$ and $U' = \langle E', Q'_P, Q'_R, pre', sub', sub'_P, sub'_C \rangle$ are bisimilar ($U \leftrightarroweq U'$) iff there exists a non-empty binary relation $F \subseteq E \times E'$ such that for all $e \in E$ and all $e' \in E'$, if eFe' then the following holds:

- $pre(e) = pre'(e')$
- $sub(e) = sub'(e')$
- $sub_P(e) = sub'_P(e')$
- $sub_C(e) = sub'_C(e')$
- for all $i \in Ag$ and all $f \in E$, if $e \rightarrow^{Q_P(i)} f$ then $\exists f' \in E'$ s.t. $e' \rightarrow^{Q'_P(i)} f'$ and fFf' (zig)
- for all $i \in Ag$ and all $f' \in E'$, if $e' \rightarrow^{Q_P(i)} f'$ then $\exists f \in E$ s.t. $e \rightarrow^{Q'_P(i)} f$ and fFf' (zag)
- above zig and zag conditions in which $Q_P(i)^{\smallsmile}$ is substituted for $Q_P(i)$
- for all $a \in Q_R$ (with corresponding $a' \in Q'_R$) and all $f \in E$, if $e \rightarrow^{a} f$ then $\exists f' \in E'$ s.t. $e' \rightarrow^{a'} f'$ and fFf' (zig)
- for all $a' \in Q'_R$ (with corresponding $a \in Q_R$) and all $f' \in E'$, if $e' \rightarrow^{a'} f'$ then $\exists f \in E$ s.t. $e \rightarrow^{a} f$ and fFf' (zag)

We say that $U, e \leftrightarroweq U', e'$ iff $U \leftrightarroweq U'$, $e \in E$, $e' \in E'$ and eFe'.

Remark 6.7. Note that our notion of bisimulation (for both models and update models) also includes zig and zag conditions for the converse of all atomic programs π. All programs that contain the converse operator can be rewritten to equivalent programs in which the converse operator is only applied to atomic programs, as axioms 7 through 10 illustrate. Since the converse operator is not safe for bisimulation, as is shown in [vB], we force bisimulation for converse programs by zig and zag conditions for converse atomic programs. All programs can be defined with atomic programs, converse atomic programs and the operators ;, $\cup$ and *, which are safe for bisimulation.

Theorem 6.8. *For all models M, w and N, v, and all formulas $\varphi \in \mathcal{L}$ holds that if $M, w \leftrightarroweq N, v$ then $w \in [\![\varphi]\!]^M$ iff $v \in [\![\varphi]\!]^N$.*

Proof. In [vBvEK] the proof of a similar result is given. In this proof a result similar to theorem 6.8 and a result similar to theorem 6.9 are simultaneously proven. Because we amend this proof for our purposes in theorem 6.9 below, the simultaneous proof can be used to prove theorem 6.8. □

Theorem 6.9. *For all models M, w and N, v, and all update models U, e holds that if $M, w \leftrightarrows N, v$ then $M \circ U, (w, e) \leftrightarrows N \circ U, (v, e)$.*

Proof. This proof amends a similar proof in [vBvEK]. In this proof theorems 6.8 and 6.9 are simultaneously proven. The proof is by induction on formulas φ and the preconditions of the relevant update models. We only amend the inductive case for the proof of theorem 6.9.

Let B be a bisimulation witnessing $M, w \leftrightarrows N, v$. Then the relation C between $W_M \times E_U$ and $W_N \times E_U$ defined by

$$(w, e)C(v, f) \text{ iff } wBv \text{ and } e = f$$

is a bisimulation. The induction hypothesis guarantees that (w, e) exists iff (v, f) exists. Suppose $(w, e)C(v, f)$. Then wBv and $e = f$. The three non-trivial checks are the checks for sameness of valuation, sameness of substitution and sameness of control. By $e = f$, we know that $sub(e) = sub(f)$. By the induction hypothesis and by the fact that w and v are bisimilar we know that $\varphi \in [\![w]\!]$ iff $\varphi \in [\![v]\!]$. Then by definition of substitutions and product update $V_{M \circ U}((w, e)) = V_{N \circ U}((v, f))$. By wBv we know that w and v fulfill the zig and zag conditions for (converse) atomic programs. Therefore, as the program constructors ;, $\cup$ and * are safe for bisimulation (see [vB]), w and v fulfill the zig and zag conditions for any program. By $e = f$ we know that $sub_P(e) = sub_P(f)$. Thus (w, e) and (v, f) also fulfill the zig and zag conditions. Furthermore, by $e = f$ we know that $sub_C(e) = sub_C(f)$ and by wBv we know that $C(w) = C(v)$. Then by definition of product update $C((w, e)) = C((v, f))$. □

Theorem 6.10. *For all models M, w and all update models U, e and U', e' holds that if $U, e \leftrightarrows U', e'$ then $M \circ U, (w, e) \leftrightarrows M \circ U', (w, e')$.*

Proof. This proof amends a similar proof in [vBvEK]. Let R be a bisimulation witnessing $U, e \leftrightarrows U', e'$. Then the relation C between $W_M \times E_U$ and $W_M \times E_{U'}$ given by

$$(w, e)C(v, f) \text{ iff } w = v \text{ and } eRf$$

is a bisimulation.

Suppose $(w, e)C(v, e')$. Then $w = v$ and eRe'. We need to check for sameness of valuation, fulfillment of the zig and zag conditions and sameness of control. By eRe' we know that

$sub(e) = sub(e')$. Then by $w = v$ we know that $\varphi \in [\![w]\!]$ iff $\varphi \in [\![v]\!]$. Then by definition of substitutions and product update $V_{M \circ U}((w, e)) = V_{M \circ U'}((w, e'))$. By eRe' we know that $sub_P(e) = sub_P(e')$. By $w = v$ we know that w and v fulfill the zig and zag conditions (for any program). Then by definition of product update we know that (w, e) and (w, e') fulfill the zig and zag conditions. By eRe' we know that $sub_C(e) = sub_C(e')$. By $w = v$ we know that $C(w) = C(v)$. Then by definition of product update $C((w, e)) = C((w, e'))$. □

Our models and update models are thus well-behaved under bisimulation.

7 Model Checking Software

Checking whether a given model satisfies a given formula is a tedious job. We can automatically perform this task by developing model checking software. We have developed such software that can check our semantic intuitions. This tool can verify whether a given formula is true in a given world in a given model, it can determine whether a given model adheres to the constraints we laid on models, it can determine whether a given update model adheres to the constraints we laid on update models, it can automatically apply product update for a given model and a given update model and it can graphically show models and update models. We will discuss this piece of software here shortly by dealing with the examples from section 4 with the software. See appendix A for the full Haskell source code containing all data structures and functions.

We use modules `ModelChecker`, `Updates` and `Display`, respectively for models and model checking, update models and product update, and displaying models and update models.

```
1 module Bankruptcy
2
3 where
4
5 import ModelChecker
6 import Updates
7 import Display
8 import List
```

We can determine whether a given model `model` satisfies a given formula `form` in a given world `w` by the following call: `isTrueAt model w form`. We can check whether a given model adheres to the constraints we laid on models by calling `checkConstraints model`. Similarly for a given update model `updateModel`: `checkUpdateConstraints updateModel`. We can determine the update product as follows: `update updateModel model`. We can display a model using the Graphviz Dot tool: `showModel model`. Again, similarly for update models: `showUpdateModel updateModel`.

The models from section 4 can be stated in terms of the data structure we defined for models. Model Γ^b_S from figure 5 is stated as `modelScared`. Model Γ^b_C from figure 6 is stated as `modelConfidence`. Model Γ^b is stated as `modelBankruptcy`. In these and further examples b is encoded as `(Q 0)`, v^i_h as `(V (10*i) h)` and other counters i^x_k for counter number x and counter value k as `(V x k)`.

```
10 {-- Auxiliary Functions --}
11
12 -- reflexive basic programs
13 reflexivePi :: BasicPi -> [state] -> [(BasicPi,state,state)]
14 reflexivePi bp ss = [ (bp,s,s) | s <- ss ]
15
16 reflexivePis :: [BasicPi] -> [state] -> [(BasicPi,state,state)]
17 reflexivePis bps ss = foldr (\y x -> x ++ (reflexivePi y ss)) [] bps
18
19 reflexiveAlpha :: BasicAlpha -> [state] -> [(BasicAlpha,state,state)]
20 reflexiveAlpha ba ss = [ (ba,s,s) | s <- ss ]
21
22 reflexiveAlphas :: [BasicAlpha] -> [state] -> [(BasicAlpha,state,state)]
23 reflexiveAlphas bas ss = foldr (\y x -> x ++ (reflexiveAlpha y ss)) [] bas
24
25 -- substitutions from all agents to one agent
26 subCWorldsFromAllTo :: [Int] -> Int -> [state] -> [(state,Agent,Agent)]
27 subCWorldsFromAllTo is j ss = [ (s,(Ag i),(Ag j)) | s <- ss, i <- is ]
28
29 -- substitution that increases a counter
30 increaseCounter :: Int -> Int -> [state] -> [(state,Prop,Form)]
31 increaseCounter max c ss = [ (s,(V c 0),Top) | s <- ss ] ++
32                            [ (s,(V c k),Prop (V c (k-1))) |
33                                                         s <- ss, k <- [1..max] ]
34
35 {-- Examples --}
36
37 -- states for the bankruptcy game
38 modelStates :: [String]
39 modelStates = ["e","c1","s1","c1c2","c1s2","s1c2","s1s2",
40               "c1c2c3","c1c2s3","c1s2c3","c1s2s3",
41               "s1c2c3","s1c2s3","s1s2c3","s1s2s3"]
42
43 -- basic pi programs for the bankruptcy game without confidence
44 scaredPi :: [(BasicPi,String,String)]
45 scaredPi =
46       [(P 2,"c1","s1"),(P 3,"c1","s1"),(P 1,"c1c2","c1s2"),
47        (P 3,"c1c2","c1s2"),(P 2,"c1c2","s1c2"),(P 3,"c1c2","s1c2"),
48        (P 1,"c1s2","s1s2"),(P 3,"s1c2","c1c2"),(P 1,"s1c2","s1s2"),
49        (P 3,"s1c2","s1s2"),(P 1,"c1c2c3","c1c2s3"),(P 2,"c1c2c3","c1c2s3"),
50        (P 3,"c1c2c3","c1s2c3"),(P 1,"c1c2s3","c1s2c3"),(P 3,"c1c2s3","c1s2s3"),
51        (P 2,"c1c2s3","s1c2c3"),(P 1,"c1s2c3","c1c2s3"),(P 1,"c1s2c3","c1s2s3"),
52        (P 2,"c1s2c3","c1s2s3"),(P 3,"c1s2c3","s1c2c3"),(P 3,"c1s2s3","s1c2s3"),
53        (P 2,"c1s2s3","s1s2c3"),(P 2,"s1c2c3","c1c2s3"),(P 3,"s1c2c3","c1s2c3"),
```

```
54         (P 1,"s1c2c3","s1c2s3"),(P 2,"s1c2c3","s1c2s3"),(P 3,"s1c2c3","s1s2c3"),
55         (P 3,"s1c2s3","c1s2s3"),(P 1,"s1c2s3","s1s2c3"),(P 3,"s1c2s3","s1s2s3"),
56         (P 2,"s1s2c3","c1s2s3"),(P 1,"s1s2c3","s1c2s3"),(P 1,"s1s2c3","s1s2s3"),
57         (P 2,"s1s2c3","s1s2s3")]
58
59  -- inverts all basic pi programs
60  invertPi :: [(BasicPi,state,state)] -> [(BasicPi,state,state)]
61  invertPi relpi = map (\(p,x,y) -> (p,y,x)) relpi
62
63  -- basic pi programs for the bankruptcy game with confidence
64  confidencePi :: [(BasicPi,String,String)]
65  confidencePi = invertPi scaredPi
66
67  constructCounter :: [Int] -> [Prop] -> state -> (state,[Prop])
68  constructCounter prefs props s = (s,(props ++ (f 1 prefs)))
69       where
70       f _ [] = []
71       f n (pr:prs) = [ (V (10*n) i) | i <- [0..pr] ] ++ (f (n+1) prs)
72
73  -- model for the bankruptcy game, given a set of pi programs
74  modelBankrupt :: [(BasicPi,String,String)] -> ActPlausM String
75  modelBankrupt rel = Mo -- states
76                        modelStates
77                        -- agents
78                        [Ag 1, Ag 2, Ag 3]
79                        -- valuation
80                        [ (constructCounter [6,6,6] [] "c1c2c3"),
81                          (constructCounter [6,6,4] [] "c1c2s3"),
82                          (constructCounter [6,4,6] [] "c1s2c3"),
83                          (constructCounter [1,4,4] [Q 0] "c1s2s3"),
84                          (constructCounter [4,6,6] [] "s1c2c3"),
85                          (constructCounter [4,1,4] [Q 0] "s1c2s3"),
86                          (constructCounter [4,4,1] [Q 0] "s1s2c3"),
87                          (constructCounter [4,4,4] [Q 0] "s1s2s3") ]
88                        -- pi relations
89                        ((reflexivePis [P 1,P 2,P 3] modelStates) ++ rel)
90                        -- alpha relations
91                        [(R 11,"e","c1"),(R 12,"e","s1"),
92                         (R 21,"c1","c1c2"),(R 22,"c1","c1s2"),
93                         (R 21,"s1","s1c2"),(R 22,"s1","s1s2"),
94                         (R 31,"c1c2","c1c2c3"),(R 32,"c1c2","c1c2s3"),
95                         (R 31,"c1s2","c1s2c3"),(R 32,"c1s2","c1s2s3"),
96                         (R 31,"s1c2","s1c2c3"),(R 32,"s1c2","s1c2s3"),
97                         (R 31,"s1s2","s1s2c3"),(R 32,"s1s2","s1s2s3")]
98                        -- control
99                        [("e",Ag 1),("c1",Ag 2),("s1",Ag 2),
100                        ("c1c2",Ag 3),("c1s2",Ag 3),("s1c2",Ag 3),("s1s2",Ag 3)]
101                       -- actual state
102                       ["e"]
103
```

```
104 -- model for the bankruptcy game without preference
105 modelBankruptcy :: ActPlausM String
106 modelBankruptcy = modelBankrupt (nub (confidencePi ++ scaredPi))
107
108 -- model for the bankruptcy game with confidence
109 modelConfidence :: ActPlausM String
110 modelConfidence = modelBankrupt confidencePi
111
112 -- model for the bankruptcy game without confidence
113 modelScared :: ActPlausM String
114 modelScared = modelBankrupt scaredPi
```

Instead of just stating the different models, we can also construct these models from scratch as in section 4.1. For a given set of agents `ags` we state model $\mathcal{B}$ as `modelB ags`. Update model $U(i)$ given an agent `i` is stated as `updateFirst ags i`. Update model F is stated as `updateSecond ags`. Update model $P(i, a, G)$ for a given action `a` (where `1` corresponds to c_i and `2` to s_i) and a given group of agents `group` corresponding to G is stated as `updateThird ags group a i`. Model Γ^b_S is constructed as `modelB3S3`, model Γ^b_C as `modelB3C3` and model Γ^b as `modelB3P`.

```
116 {-- Constructing the model by updates --}
117
118 -- basic model
119 modelB :: [Int] -> ActPlausM Int
120 modelB ags = let agents = [ (Ag i) | i <- ags ] in
121              Mo -- states
122                 [0]
123                 -- agents
124                 agents
125                 -- val
126                 [(0,[Q 0])]
127                 -- pi
128                 (reflexivePis [ (P i) | i <- ags ] [0])
129                 -- alpha
130                 ([ (R (10*i+1),0,0) | i <- ags ] ++
131                  [ (R (10*i+2),0,0) | i <- ags ])
132                 -- control
133                 []
134                 -- dist
135                 [0]
136
137 -- basic model with three agents
138 modelB3 :: ActPlausM Int
139 modelB3 = modelB [1,2,3]
140
141 -- obtains pi program substitutions that are safe for updating
142 safeSubstitutions :: [state] -> [Int] -> (Int -> Pi) -> [(state,BasicPi,Pi)]
143 safeSubstitutions ss bps fp =
144       [ (s,(P bp), (PUnion (PConcat (PTest (Prop (C bp))) (PConcat (fp bp)
```

```
145                             (PTest (Prop (C bp))) ))
146                             (PConcat (PTest (Neg (Prop (C bp)))) (PConcat (fp bp)
147                             (PTest (Neg (Prop (C bp)))) ))  ))
148                                                                 | s <- ss, bp <- bps ]
149
150 -- update that introduces a choice for a given agent
151 updateFirst :: [Int] -> Int -> UpdateM Int
152 updateFirst ags j = Up -- events
153                        [1,2,3,4]
154                        -- agents
155                        [ (Ag i) | i <- ags ]
156                        -- pi relations
157                        ((reflexivePis [ (P i) | i <- ags ] [1,2,3,4] ) ++
158                          [ (P i,3,4) | i <- ags, i /= j ] ++
159                          [ (P i,4,3) | i <- ags, i /= j ] ++
160                          [ (P i,1,2) | i <- ags] ++
161                          [ (P i,2,1) | i <- ags])
162                        -- alpha relations
163                        ((reflexiveAlphas [(R (10*i+1)) | i <- ags] [1,3,4]) ++
164                          (reflexiveAlphas [(R (10*i+2)) | i <- ags] [1,3,4]) ++
165                          [ (R (10*i+1),1,2) | i <- ags ] ++
166                          [ (R (10*i+2),1,2) | i <- ags ] ++
167                          [(R (10*j+1),2,3),(R (10*j+2),2,4)])
168                        -- pre
169                        [(1,Neg (Prop (Q 0))),(2,Prop (Q 0)),
170                         (3,Prop (Q 0)),(4,Prop (Q 0))]
171                        -- sub
172                        ((increaseCounter 5 1 [3]) ++
173                          (increaseCounter 5 2 [4]) ++
174                          [(1,Q (10*j+1),Neg (Top)),(1,Q (10*j+2),Neg (Top)),
175                          (2,Q 0,Neg Top),(2,Q (10*j+1),Neg (Top)),
176                           (2,Q (10*j+2),Neg (Top)),
177                           (3,Q (10*j+1),Top),(3,Q (10*j+2),Neg (Top)),
178                          (4,Q (10*j+1),Neg (Top)),(4,Q (10*j+2),Top)])
179                        -- subp
180                        (safeSubstitutions [1] ags (\i -> (PBasic (P i))) )
181                        -- subc
182                        ((subCWorldsFromAllTo ags 0 [2,3,4]) ++
183                          [(2,Ag 0,Ag j)])
184                        -- dist
185                        [1,2]
186
187 -- update that finishes the model after introducing choices
188 updateSecond :: Int -> [Int] -> UpdateM Int
189 updateSecond max ags =
190   let b = Disj (map (\k -> Conj [Prop (V 2 k), Neg (Prop (V 1 k))]) [1..max]) in
191                       Up -- events
192                       [5,6]
193                       -- agents
194                       [ (Ag i) | i <- ags ]
```

```
195                          -- pi relations
196                          ((reflexivePis [ (P i) | i <- ags ] [5,6]) ++
197                            ([ (P i,5,6) | i <- ags ] ++ [ (P i,6,5) | i <- ags ]))
198                          -- alpha relations
199                          ((reflexiveAlphas [ (R (10*i+1)) | i <- ags ] [5]) ++
200                            (reflexiveAlphas [ (R (10*i+2)) | i <- ags ] [5]) ++
201                            [ (R (10*i+1),5,6) | i <- ags ] ++
202                            [ (R (10*i+2),5,6) | i <- ags ])
203                          -- pre
204                          [(5,Neg (Prop (Q 0))),(6,Prop (Q 0))]
205                          -- sub
206                          ([(6,Q 0,Neg Top), (6,Q 1,b)] ++
207                            [ (6,V (10*i) n,Disj [Prop (Q (10*i+2)),
208                                           Conj [Prop (Q (10*i+1)),Neg b]])
209                                                          | i <- ags, n <- [2..4]] ++
210                            [ (6,V (10*i) m,Conj [Prop (Q (10*i+1)),Neg b])
211                                                          | i <- ags, m <- [5,6] ] ++
212                            [ (6,V (10*i) 1,Top) | i <- ags ])
213                          -- subp
214                          (safeSubstitutions [5] ags (\i -> (PBasic (P i))) )
215                          -- subc
216                          (subCWorldsFromAllTo ags 0 [6])
217                          -- dist
218                          [5]
219
220   -- update that introduces beliefs (for a given group of agents)
221   -- about the action of a given agent
222   updateThird :: [Int] -> [Int] -> Int -> Int -> UpdateM Int
223   updateThird ags group act ag = -- act == 1 corresponds to confidence,
224                                  -- act == 2 corresponds to scare
225                                  Up -- events
226                                  [7,8,9]
227                                  -- agents
228                                  [ (Ag i) | i <- ags ]
229                                  -- pi relations
230                                  ((reflexivePis [ (P i) | i <- ags ] [7,8,9]) ++
231                                    ([ (P g,(10-act),(7+act))  | g <- group ]) ++
232                                    ([ (P i,7,8) | i <- ags ]) ++
233                                    ([ (P i,8,7) | i <- ags ]) ++
234                                    ([ (P i,7,9) | i <- ags ]) ++
235                                    ([ (P i,9,7) | i <- ags ]))
236                                  -- alpha relations
237                                  ((reflexiveAlphas [ (R (10*i+1)) | i <- ags ]
238                                      [7,8,9]) ++
239                                    (reflexiveAlphas [ (R (10*i+2)) | i <- ags ]
240                                      [7,8,9]) ++
241                                    ([(R (10*ag+1),7,8),(R (10*ag+2),7,9)]))
242                                  -- pre
243                                  [ (7,Conj [Neg (Prop (Q (10*ag+1))),
244                                                    Neg (Prop (Q (10*ag+2)))]),
```

```
245                                  (8,Prop (Q (10*ag+1))),
246                                                    (9,Prop (Q (10*ag+2))) ]
247                                -- sub
248                                []
249                                -- subp
250                                (safeSubstitutions [7,8,9] ags
251                                   (\i -> (PBasic (P i))) )
252                                -- subc
253                                []
254                                -- dist
255                                [7]
256
257 -- examples built from scratch
258 modelB31 = (update (updateFirst [1,2,3] 1) modelB3)
259 modelB32 = (update (updateFirst [1,2,3] 2) modelB31)
260 modelB33 = (update (updateFirst [1,2,3] 3) modelB32)
261
262 -- model for the bankruptcy game with three players
263 modelB3P = (update (updateSecond 5 [1,2,3]) modelB33)
264
265 modelB3C1 = (update (updateThird [1,2,3] [1,2,3] 1 1) modelB3P)
266 modelB3C2 = (update (updateThird [1,2,3] [1,2,3] 1 2) modelB3C1)
267 -- model for the confidence bankruptcy game with three players
268 modelB3C3 = (update (updateThird [1,2,3] [1,2,3] 1 3) modelB3C2)
269
270 modelB3S1 = (update (updateThird [1,2,3] [1,2,3] 2 1) modelB3P)
271 modelB3S2 = (update (updateThird [1,2,3] [1,2,3] 2 2) modelB3S1)
272 -- model for the scared bankruptcy game with three players
273 modelB3S3 = (update (updateThird [1,2,3] [1,2,3] 2 3) modelB3S2)
```

Furthermore, the update models from section 4.4 can be implemented as well. We state U_1 as `turningOne ags` for a given set of agents `ags`. Update model U_2 is stated as `turningTwo ags`.

```
275 -- update that turns the tide by reversing beliefs
276 turningOne :: [Int] -> UpdateM Int
277 turningOne ags = Up -- events
278                     [0]
279                     -- agents
280                     [ (Ag i) | i <- ags ]
281                     -- pi relations
282                     (reflexivePis [ (P i) | i <- ags ] [0])
283                     -- alpha relations
284                     ((reflexiveAlphas [ (R (10*i+1)) | i <- ags ] [0]) ++
285                       (reflexiveAlphas [ (R (10*i+2)) | i <- ags ] [0]))
286                     -- pre
287                     []
288                     -- sub
289                     []
290                     -- subp
291                     (safeSubstitutions [0] ags (\i -> (PConv (PBasic (P i)))) )
```

```
292                         -- subc
293                         []
294                         -- dist
295                         [0]
296
297 -- update that turns the tide by changing preferences over outcomes
298 turningTwo :: [Int] -> UpdateM Int
299 turningTwo ags = Up -- events
300                         [0]
301                         -- agents
302                         [ (Ag i) | i <- ags ]
303                         -- pi relations
304                         (reflexivePis [ (P i) | i <- ags ] [0])
305                         -- alpha relations
306                         ((reflexiveAlphas [ (R (10*i+1)) | i <- ags ] [0]) ++
307                           (reflexiveAlphas [ (R (10*i+2)) | i <- ags ] [0]))
308                         -- pre
309                         []
310                         -- sub
311                         [ (0,V (10*i) n, Disj [Conj [(Prop (Q (10*i+1))),
312                                                    (Prop (Q 1))],Prop (V (10*i) n)])
313                                                                | i <- ags, n <- [1..5] ]
314                         -- subp
315                         (safeSubstitutions [0] ags (\i -> (PBasic (P i))) )
316                         -- subc
317                         []
318                         -- dist
319                         [0]
```

We also implemented the formulas and programs from section 4.3. Formula τ is stated as `tau ags` for a given set of agents `ags`. The weak belief program $>_i^{max}$ is stated as `piMax i`. The unfolding of the program *opt* to a certain level `n` μ^n, where the counters have a maximum `m`, is stated as `opt ags m n`. The formula $opt(i, a)$ for a given agent `i` and a given action `a` is stated as `optimal ags m n i a`.

```
321 {-- Analysis --}
322
323 -- formula that is true in all and only in terminal worlds
324 tau :: [Int] -> Form
325 tau (ag:ags) = AlphaBox (AUnion (foldl (\x y -> AUnion x y) (ABasic (R (10*ag+1)))
326                          (map (\a -> ABasic (R (10*a+1))) ags))
327                          (foldl (\x y -> AUnion x y) (ABasic (R (10*ag+2)))
328                          (map (\a -> ABasic (R (10*a+2))) ags))) (Neg Top)
329
330 -- union of alpha programs
331 unite :: [a] -> (a -> Alpha) -> Alpha
332 unite (x:xs) f = foldl (\y z -> AUnion y z) (f x) (map f xs)
333
334 -- concatenation of alpha programs
335 concatenate :: [a] -> (a -> Alpha) -> Alpha
```

```
336 concatenate (x:xs) f = foldl (\y z -> AConcat y z) (f x) (map f xs)
337
338 -- disjunction of formulas
339 disjunction :: [a] -> (a -> Form) -> Form
340 disjunction xs f = Disj (map f xs)
341
342 -- conjunction of formulas
343 conjunction :: [a] -> (a -> Form) -> Form
344 conjunction xs f = Conj (map f xs)
345
346 -- weak belief relation
347 piMax :: Int -> Pi
348 piMax i = PConcat ({-PKleene-} (PBasic (P i))) (PTest (PiBox (PBasic (PS i))
349                                                                       (Neg Top)))
350
351 -- optimal action alpha program
352 opt :: [Int] -> Int -> Int -> Alpha
353 opt ags _ 0 = ATest (tau ags)
354 opt ags max n =
355             AUnion (ATest (tau ags))
356             (AConcat (ATest (Neg (tau ags)))
357             (unite ags (\i -> AConcat (ATest (Prop (C i))) (unite [1..max] (\h ->
358             (unite [ ABasic (R (10*i+1)), ABasic (R (10*i+2)) ] (\a ->
359             (AConcat (ATest (PiBox (piMax i) (AlphaBox (AConcat (a)
360                  (opt ags max (n-1))) (Prop (V (10*i) h)))))
361             (AConcat (concatenate [ c | c <- [ ABasic (R (10*i+1)),
362                  ABasic (R (10*i+2)) ], c /= a] (\b ->
363             (ATest (PiBox (piMax i) (AlphaBox (AConcat (b) (opt ags max (n-1)))
364                  (Neg (Prop (V (10*i) (h+1))))))) ))
365             (AConcat (a) (opt ags max (n-1)))
366             ))
367             )))))))
368
369 -- formula that checks whether a given action is optimal for a given agent
370 optimal :: [Int] -> Int -> Int -> Int -> BasicAlpha -> Form
371 optimal ags max n i a =
372           Conj [Neg (tau ags),
373           Prop (C i),
374           Neg (AlphaBox (ABasic a) (Neg Top)),
375           disjunction [1..max] (\h ->
376             Conj [PiBox (piMax i) (AlphaBox (AConcat (ABasic a) (opt ags max n))
377                  (Prop (V (10*i) h))),
378             conjunction [ c | c <- [ (R (10*i+1)), (R (10*i+2)) ], c /= a ] (\b ->
379               PiBox (piMax i) (AlphaBox (AConcat (ABasic b) (opt ags max n))
380                    (Neg (Prop (V (10*1) (h+1)))))
381             )]
382           )]
```

Note that in line 348 `PKleene` is commented out. In general this Kleene closure is needed, but our example is constructed in such a fashion that the closure operator can be left out

in this case. In our example models, namely, there are no P_i paths between two worlds that aren't directly connected. Leaving out the closure operator reduces complexity of checking whether the formula `optimal` is true in a given model.

Having implemented all this machinery, we are now ready to verify whether, for instance, action c_2 is the optimal action for agent 2 in Γ_C^b in world c_1. We use model `modelB3C3`, world `(((((((0,3),2),1),5),8),7),7)` (that incidently corresponds to c_1) and formula `(optimal [1,2,3] 6 2 2 (R 21))`. We see that, indeed,

```
isTrueAt modelB3C3 (((((((0,3),2),1),5),8),7),7) (optimal [1,2,3] 6 2 2 (R 21))
```

evaluates as `True`. Conversely,

```
isTrueAt modelB3C3 (((((((0,3),2),1),5),8),7),7) (optimal [1,2,3] 6 2 2 (R 22))
```

evaluates as `False`.

7.1 Possible Improvements

Our model checking software is implemented straightforwardly and relatively naively. Sets are implemented as ordered lists without duplicates. Relations are implemented as functions. All information about models and update models is implemented as lists (of pairs). Because of the way lists are implemented in Haskell this is not the most efficient implementation. Checking of certain properties for larger models can thus get quite time-consuming. Even for our examples, checking whether `checkConstraints modelB3S3` or whether `isTrueAt modelB3C3 (((((((0,2),1),1),5),7),7),7) (optimal [1,2,3] 6 3 1 (R 11))` already takes an intractable amount of time. One of the aspects of our model checking software that can be improved is thus efficiency.

Another possible improvement to our model checking software is adding certain functionality. An example of such functionality is model minimization under bisimulation as in [vE$_1$]. Our software implementation comprises merely a certain basic functionality. Many more interesting features could be added to our software.

8 Conclusion

In this thesis we developed a logic that makes the explicit analysis of belief in social software possible. This logic consists of a static logic, based on propositional dynamic logic, and product updates. We showed how the example of a bank's going bankrupt due to its clients' beliefs can be analyzed using our logic. Furthermore, we proved certain formal properties of our logic. It has a complete axiomatization. The updates are reducible to the static

logic. The updates are well-behaved with respect to bisimulation. We also gave a software implementation for model checking.

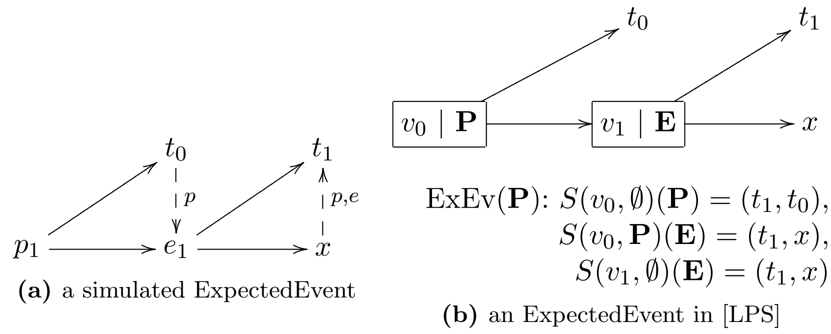
8.1 Further Research

One obvious direction for further research is the application of the logic we developed to the analysis of more social situations and procedures in which belief is essential to the analysis. We have only used the logic for the analysis of one example. There are many other procedures that can be analyzed using our logic.

Another direction for further research is the investigation of the connection between our logic and other logics that are used to analyze social software. An example of such an other logic used for social software is the history-based epistemic logic that Pacuit uses in [P_1] to analyze knowledge-based obligation. We can perform the analyses in [P_1] in our logic as well. In order to do so, we can make the following construction. Given the protocol, we can construct an α-program tree that corresponds to the protocol (i.e., for any history in the protocol, there is a path in our tree that corresponds to this history). Then we can define equivalence relations P_i as follows: for all states s and s', corresponding to finite prefixes H and H' respectively, we let $s \rightarrow^{P_i} s'$ iff $\lambda_i(H) = \lambda_i(H')$. This way we can model history-based runs in our logic. Further research is needed to establish how exactly the analysis in [P_1] can be translated to our logic.

The application of our logic to other domains than social software is another direction for further research that could be quite fruitful. An example of such an application is the following. Löwe, Pacuit and Saraf analyze the structure of stories using games with beliefs in [LPS]. Their formalization can be simulated and generalized using our logic. We give an example to illustrate the translation from their models to our models. In figure 9 we can see how we can model a game from [LPS] corresponding to an unexpected event. We construct a α-program tree corresponding to the form of the game. We can also simulate the (partial) states S as π-programs straightforwardly. As soon as we translated the game to our models, we have all the freedom to generalize the approach from [LPS]. Further research is needed to establish how exactly the analysis in [LPS] can be performed and generalized using our logic.

Further research thus lies in the application of our logic to both similar analyses and different domains, as well as in the investigation of the connection between our logic and other logics.



(a) a simulated ExpectedEvent

(b) an ExpectedEvent in [LPS]

Figure 9: An example of a simulation of a game from [LPS].

References

[BdRV] Patrick Blackburn, Maarten de Rijke, and Yde Venema, *Modal logic*, Cambridge University Press, New York, 2001.

[DvE] Kees Doets and Jan van Eijck, *The haskell road to logic, maths and programming*, Texts in Computing, King's College Publications, London, 2004.

[FL] Michael J. Fischer and Richard E. Ladner, *Propositional dynamic logic of regular programs*, Journal of Computer and System Sciences **18** (1979), no. 2, 194–211.

[KP] Dexter Kozen and Rohit Parikh, *An elementary proof of the completeness of PDL*, Theoretical Computer Science **14** (1981), 113–118.

[LPS] Benedikt Löwe, Eric Pacuit, and Sanchit Saraf, *Analyzing stories as games with changing and mistaken beliefs*, Tech. report, Institute of Logic, Language and Computation, 2008.

[P_1] Eric Pacuit, *Topics in social software: Information in strategic situations*, Ph.D. thesis, City University of New York Graduate Center, 2005.

[P_2] Rohit Parikh, *The completeness of propositional dynamic logic*, MFCS (Józef Winkowski, ed.), Lecture Notes in Computer Science, **64**, Springer, 1978, pp. 403–415.

[P_3] Rohit Parikh, *Social software*, Synthese **132** (2002), no. 3, 187–211.

[PP] Eric Pacuit and Rohit Parikh, *Social interaction, knowledge and social software*, Interactive Computation: A New Paradigm (Dina Goldin, Scott Smolka, and Peter Wagner, eds.), 2006.

[vB] Johan van Benthem, *Program constructions that are safe for bisimulation*, Studia Logica **60** (1998), no. 2, 311–330.

[vBvEK] Johan van Benthem, Jan van Eijck, and Barteld P. Kooi, *Logics of communication and change*, Inf. Comput. **204** (2006), no. 11, 1620–1662.

[vE_1] Jan van Eijck, *Interactive logic – proceedings of the 7th augustus de morgan workshop*, DEMO – A Demo of Epistemic Modelling (Johan van Benthem, Dov Gabbay, and Benedikt Löwe, eds.), Amsterdam University Press, 2007, pp. 305–363.

[vE_2] Jan van Eijck, *Yet more modal logics of preference change and belief revision*, New Perspectives on Games and Interaction (K.R. Apt and R. van Rooij, eds.), Amsterdam University Press, 2008, pp. 81–104.

[vEW] Jan van Eijck and Yanjing Wang, *Propositional dynamic logic as a logic of belief revision*, WoLLIC '08: Proceedings of the 15th international workshop on Logic, Language, Information and Computation (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 136–148.

A Model Checking Software Source

The source code is divided into several modules. We will discuss these modules, and give the source code of the modules.

A.1 SetOrd

The module *SetOrd*, which originates from [DvE], contains functions and data structures that implement sets as ordered lists without duplicates. Comments are left out, for the naming of the functions is informative enough.

```
module SetOrd (Set(..),emptySet,isEmpty,inSet,subSet,insertSet,
               deleteSet,powerSet,takeSet,(!!!),list2set)

where

import List (sort)

{-- Sets implemented as ordered lists without duplicates --}

newtype Set a = Set [a] deriving (Eq,Ord)

instance (Show a) => Show (Set a) where
    showsPrec _ (Set s) str = showSet s str

showSet []     str = showString "{}" str
showSet (x:xs) str = showChar '{' ( shows x ( showl xs str))
     where showl []     str = showChar '}' str
           showl (x:xs) str = showChar ',' (shows x (showl xs str))

emptySet  :: Set a
emptySet = Set []

isEmpty  :: Set a -> Bool
isEmpty (Set []) = True
isEmpty _        = False
```

```
26
27 inSet  :: (Ord a) => a -> Set a -> Bool
28 inSet x (Set s) = elem x (takeWhile (<= x) s)
29
30 subSet :: (Ord a) => Set a -> Set a -> Bool
31 subSet (Set []) _       = True
32 subSet (Set (x:xs)) set = (inSet x set) && subSet (Set xs) set
33
34 insertSet :: (Ord a) => a -> Set a -> Set a
35 insertSet x (Set s) = Set (insertList x s)
36
37 insertList x [] = [x]
38 insertList x ys@(y:ys2) = case compare x y of
39                                 GT -> y : insertList x ys2
40                                 EQ -> ys
41                                 _  -> x : ys
42
43 deleteSet :: Ord a => a -> Set a -> Set a
44 deleteSet x (Set s) = Set (deleteList x s)
45
46 deleteList x [] = []
47 deleteList x ys@(y:ys2) = case compare x y of
48                                 GT -> y : deleteList x ys2
49                                 EQ -> ys2
50                                 _  -> ys
51
52 list2set :: Ord a => [a] -> Set a
53 list2set [] = Set []
54 list2set (x:xs) = insertSet x (list2set xs)
55
56 powerSet :: Ord a => Set a -> Set (Set a)
57 powerSet (Set xs) =
58    Set (sort (map (\xs -> (list2set xs)) (powerList xs)))
59
60 powerList  :: [a] -> [[a]]
61 powerList  [] = [[]]
62 powerList  (x:xs) = (powerList xs)
63                      ++ (map (x:) (powerList xs))
64
```

```
takeSet :: Eq a => Int -> Set a -> Set a
takeSet n (Set xs) = Set (take n xs)

infixl 9 !!!

(!!!) :: Eq a => Set a -> Int -> a
(Set xs) !!! n = xs !! n
```

A.2 Rel

The module *Rel*, which also originates from [DvE], contains functions and data structures that implement relations. Again, comments are left out, for the naming of the functions is informative enough.

```
module Rel

where

import List
import SetOrd

{-- Relations implemented as functions --}

type Rel a = a -> a -> Bool

eq :: Eq a => (a,a) -> Bool
eq = uncurry (==)

lessEq :: Ord a => (a,a) -> Bool
lessEq = uncurry (<=)

inverse :: ((a,b) -> c) -> ((b,a) -> c)
inverse f (x,y) = f (y,x)

emptyR :: Rel a
emptyR = \ _ _ -> False

```

```haskell
24 list2Rel :: Eq a => [(a,a)] -> Rel a
25 list2Rel xys = \ x y -> elem (x,y) xys
26
27 idR :: Eq a => [a] -> Rel a
28 idR xs = \ x y -> x == y && elem x xs
29
30 invR :: Rel a -> Rel a
31 invR = flip
32
33 inR :: Rel a -> (a,a) -> Bool
34 inR = uncurry
35
36 reflR :: Set a -> Rel a -> Bool
37 reflR (Set xs) r = and [ r x x | x <- xs ]
38
39 irreflR :: Set a -> Rel a -> Bool
40 irreflR (Set xs) r = and [ not (r x x) | x <- xs ]
41
42 symR :: Set a -> Rel a -> Bool
43 symR (Set xs) r = and [ not (r x y && not (r y x)) | x <- xs, y <- xs ]
44
45 transR :: Set a -> Rel a -> Bool
46 transR (Set xs) r = and
47               [ not (r x y && r y z && not (r x z))
48                         | x <- xs, y <- xs, z <- xs ]
49
50 unionR :: Rel a -> Rel a -> Rel a
51 unionR r s x y = r x y  || s x y
52
53 intersR :: Rel a -> Rel a -> Rel a
54 intersR r s x y = r x y && s x y
55
56 reflClosure :: Eq a => Rel a -> Rel a
57 reflClosure r = unionR r (==)
58
59 symClosure :: Rel a -> Rel a
60 symClosure r = unionR r (invR r)
61
62 transClosure :: Eq a => [a] -> Rel a -> Rel a
```

```
63 transClosure xs r = f (length xs)
64   where f 0 = (idR xs)
65         f n = unionR (repeatR xs r n) (f (n-1))
66
67 compR :: [a] -> Rel a -> Rel a -> Rel a
68 compR xs r s x y = or [ r x z && s z y | z <- xs ]
69
70 repeatR :: [a] -> Rel a -> Int -> Rel a
71 repeatR xs r n | n < 1      = error "argument < 1"
72                | n == 1     = r
73                | otherwise = compR xs r (repeatR xs r (n-1))
74
75 equivalenceR :: Ord a => Set a -> Rel a -> Bool
76 equivalenceR set r = reflR set r && symR set r && transR set r
```

A.3 ModelChecker

The module *ModelChecker* is the main module of the program. It contains data structures that represent modules and functions that implement model checking.

```
1  module ModelChecker
2  where
3
4  import List
5  import SetOrd
6  import Rel
7
8  {-- Data Structures --}
9
10 -- agents
11 data Agent = Ag Int deriving (Eq,Ord)
12
13 instance Show Agent where
14   show (Ag i) = "ag" ++ show i
15
16 -- propositions (basic propositions q, control propositions c, counters v)
```

```
17 data Prop = Q Int
18           | C Int
19           | V Int Int
20           deriving (Eq,Ord)
21
22 instance Show Prop where
23   show (Q i) = 'q':show i
24   show (C i) = 'c':show i
25   show (V i j) = 'v':show i ++ "-" ++ show j
26
27 -- basic pi programs
28 data BasicPi = P Int
29              | PS Int
30              deriving (Eq,Ord)
31
32 instance Show BasicPi where
33   show (P i) = 'P':show i
34   show (PS i) = 'P':'S':show i
35
36 -- basic alpha programs
37 data BasicAlpha = R Int
38                 deriving (Eq,Ord)
39
40 instance Show BasicAlpha where
41   show (R i) = 'R':show i
42
43 -- complex pi programs
44 data Pi = PBasic BasicPi
45         | PConcat Pi Pi
46         | PUnion Pi Pi
47         | PKleene Pi
48         | PConv Pi
49         | PTest Form
50         deriving (Eq,Ord)
51
52 instance Show Pi where
53   show (PBasic basicpi)      = show basicpi
54   show (PConcat pi1 pi2)     = '(':show pi1 ++ ";" ++ show pi2 ++ ")"
55   show (PUnion pi1 pi2)      = '(':show pi1 ++ "U" ++ show pi2 ++ ")"
```

```haskell
56    show (PKleene pi1)          = show pi1 ++ "*"
57    show (PConv pi1)            = show pi1 ++ "^"
58    show (PTest form)           = '?':show form
59
60  -- complex alpha programs
61  data Alpha = ABasic BasicAlpha
62             | AConcat Alpha Alpha
63             | AUnion Alpha Alpha
64             | AKleene Alpha
65             | ATest Form
66             deriving (Eq,Ord)
67
68  instance Show Alpha where
69    show (ABasic basicalpha)        = show basicalpha
70    show (AConcat alpha1 alpha2)    = '(':show alpha1 ++ ";" ++ show alpha2 ++ ")"
71    show (AUnion alpha1 alpha2)     = '(':show alpha1 ++ "U" ++ show alpha2 ++ ")"
72    show (AKleene alpha1)           = show alpha1 ++ "*"
73    show (ATest form)               = '?':show form
74
75  -- formulas
76  data Form = Top
77            | Prop Prop
78            | Neg Form
79            | Conj [Form]
80            | Disj [Form]
81            | PiBox Pi Form
82            | AlphaBox Alpha Form
83            deriving (Eq,Ord)
84
85  instance Show Form where
86    show Top            = "T"
87    show (Neg Top)      = "F"
88    show (Prop p)       = show p
89    show (Neg f)        = '-':(show f)
90    show (Conj fs)      = '&':show fs
91    show (Disj fs)      = 'v':show fs
92    show (PiBox pi1 form)    = '{':show pi1 ++ "}" ++ show form
93    show (AlphaBox alpha1 form)    = '{':show alpha1 ++ "}" ++ show form
94
```

```haskell
95  -- models (Mo states agents val pi alpha control dist)
96  data ActPlausM state = Mo
97               [state]
98               [Agent]
99               [(state,[Prop])]
100              [(BasicPi,state,state)]
101              [(BasicAlpha,state,state)]
102              [(state,Agent)]
103              [state]  deriving (Eq,Show)
104
105 {-- Auxiliary Functions --}
106
107 implies :: Bool -> Bool -> Bool
108 implies b1 b2 = (not b1) || b2
109
110 forall :: [a] -> (a -> Bool) -> Bool
111 forall xs f = and (map f xs)
112
113 exists :: [a] -> (a -> Bool) -> Bool
114 exists xs f = or (map f xs)
115
116 dom :: ActPlausM a -> [a]
117 dom (Mo states _ _ _ _ _ _) = states
118
119 getPi :: ActPlausM a -> [(BasicPi,a,a)]
120 getPi (Mo _ _ _ rels _ _ _) = rels
121
122 getAlpha :: ActPlausM a -> [(BasicAlpha,a,a)]
123 getAlpha (Mo _ _ _ _ rels _ _) = rels
124
125 getAgent :: Agent -> Int
126 getAgent (Ag i) = i
127
128 -- obtains a list of all the different basic alpha programs
129 differentAlpha :: [(BasicAlpha,state,state)] -> [BasicAlpha]
130 differentAlpha rel = nub (map (\(x,_,_) -> x) rel)
131
132 -- converts a pi program into the program corresponding to the equivalence relation
133 getEquiv :: Pi -> Pi
```

```
134 getEquiv p = PKleene (PUnion (p) (PConv (p)) )
135
136 {-- Model Checking Functions --}
137
138 -- obtains the agent that controls a state
139 owner :: Eq state => state -> [(state,Agent)] -> Agent
140 owner s c = case find (\(w,a) -> w == s) c of
141      Just (_,a) -> a
142      Nothing -> Ag 0
143
144 -- obtains the relation for a basic pi program
145 basicPi :: Ord state => [(BasicPi,state,state)] -> BasicPi -> Rel state
146 basicPi rels pi1 = (\x y -> case find (\(pi2,w,z) -> (pi2 == pi1) && (w == x) && (z == y)) rels of
147                           Just _ -> True
148                           Nothing -> False)
149
150 -- obtains the strict relation for a basic pi program
151 basicPiS :: Ord state => [(BasicPi,state,state)] -> BasicPi -> Rel state
152 basicPiS rels pi1 = (\x y -> case find (\(pi2,w,z) -> (pi2 == pi1) && (w == x) && (z == y)) rels of
153                           Just _ -> case find (\(pi3,u,v) -> (pi3 == pi1) && (u == y) && (v == x)) rels of
154                                Just _ -> False
155                                Nothing -> True
156                           Nothing -> False)
157
158 -- obtains the relations for a complex pi program
159 obtainPi :: Ord state => ActPlausM state -> Pi -> Rel state
160 obtainPi m (PBasic pi1@(P _)) = basicPi (getPi m) pi1
161 obtainPi m (PBasic pi1@(PS x)) = basicPiS (getPi m) (P x)
162 obtainPi m (PConcat pi1 pi2) = compR (dom m) (obtainPi m pi1) (obtainPi m pi2)
163 obtainPi m (PUnion pi1 pi2) = unionR (obtainPi m pi1) (obtainPi m pi2)
164 obtainPi m (PKleene pi1) = transClosure (dom m) (obtainPi m pi1)
165 obtainPi m (PConv pi1) = (\x y -> (obtainPi m pi1) y x)
166 obtainPi m (PTest form) = (\x y -> (x == y) && (isTrueAt m x form))
167
168 -- obtains the relation for a basic alpha program
169 basicAlpha :: Ord state => [(BasicAlpha,state,state)] -> BasicAlpha -> Rel state
170 basicAlpha rels pi1 = (\x y -> case find (\(pi2,w,z) -> (pi2 == pi1) && (w == x) && (z == y)) rels of
171                           Just _ -> True
172                           Nothing -> False)
```

```
173
174 -- obtains the relation for a complex alpha program
175 obtainAlpha :: Ord state => ActPlausM state -> Alpha -> Rel state
176 obtainAlpha m (ABasic pi1) = basicAlpha (getAlpha m) pi1
177 obtainAlpha m (AConcat pi1 pi2) = compR (dom m) (obtainAlpha m pi1) (obtainAlpha m pi2)
178 obtainAlpha m (AUnion pi1 pi2) = unionR (obtainAlpha m pi1) (obtainAlpha m pi2)
179 obtainAlpha m (AKleene pi1) = transClosure (dom m) (obtainAlpha m pi1)
180 obtainAlpha m (ATest form) = (\x y -> (x == y) && (isTrueAt m x form))
181
182 -- determines whether a given formula is true in a given state in the model
183 isTrueAt :: Ord state => ActPlausM state -> state -> Form -> Bool
184 isTrueAt m w Top = True
185 isTrueAt m@(Mo _ _ _ _ _ control _) w (Prop p@(C i)) =
186   case find (\(s,ag) -> s == w) control of
187     Nothing         -> i == 0
188     Just (s,(Ag j)) -> j == i
189 isTrueAt m@(Mo _ _ val _ _ _ _) w (Prop p) =
190   case find (\(x,ps) -> x == w) val of
191     Nothing     -> False
192     Just (x,ps) -> elem p ps
193 isTrueAt m w (Neg f)    = not (isTrueAt m w f)
194 isTrueAt m w (Conj fs) = and (map (isTrueAt m w) fs)
195 isTrueAt m w (Disj fs) = or (map (isTrueAt m w) fs)
196 isTrueAt m@(Mo worlds _ _ _ _ _ _) w (PiBox pi1 f) = all (\v -> ((obtainPi m pi1) w v)
197                                                           `implies` (isTrueAt m v f)) worlds
198 isTrueAt m@(Mo worlds _ _ _ _ _ _) w (AlphaBox pi1 f) = all (\v -> ((obtainAlpha m pi1) w v)
199                                                           `implies` (isTrueAt m v f)) worlds
200
201 -- checks whether a model does not violate the constraints
202 checkConstraints :: Eq state => Ord state => ActPlausM state -> Bool
203 checkConstraints m@(Mo states agents val relpi relalpha control dist) = and [c1,c2,c3,c4]
204     where
205     -- constraint of reflexivity
206     c1 = forall agents (\(Ag a) -> forall states (\s -> elem (P a, s, s) relpi))
207     -- constraint of distinction
208     c2 = forall states (\s1 -> forall states (\s2 ->
209           let ownerS1 = owner s1 control in
210           let ownerS2 = owner s2 control in
211           ((ownerS1 /= ownerS2)
```

```
212             && (ownerS1 /= (Ag 0)))
213             `implies` (not (inR (obtainPi m (getEquiv (PBasic (P (getAgent ownerS1)))))
214                         (s1,s2) )) ))
215       -- constraint of awareness
216       c3 =  forall states (\s1 -> forall states (\s2 ->
217             forall (differentAlpha relalpha) (\a ->
218             let ownerS1 = owner s1 control in
219             let ownerS2 = owner s2 control in
220             and [(ownerS1 == ownerS2),
221             (inR (obtainPi m (getEquiv (PBasic (P (getAgent ownerS1)))))
222                         (s1,s2) ),
223             (exists states (\s3 -> (a,s1,s3) `elem` relalpha))
224             ]
225             `implies` (exists states (\s4 -> (a,s2,s4) `elem` relalpha)) )))
226       -- constraint of nondeterminism
227       c4 = forall states (\s1 -> forall states (\s2 -> forall states (\s3 ->
228             forall (differentAlpha relalpha) (\a ->
229             let ownerS1 = owner s1 control in
230             and [((a,s1,s2) `elem` relalpha),((a,s1,s3) `elem` relalpha)]
231             `implies` (inR (obtainPi m (getEquiv (PBasic (P (getAgent ownerS1)))))
232                         (s2,s3) ) ))))
```


A.4 Updates

The module *Updates* contains data structures that represent update models and functions that implement product update.

```
1 module Updates
2 where
3
4 import List
5 import SetOrd
6 import Rel
7 import ModelChecker
8
9 {-- Data Structures --}
```

```
10
11  -- update models (Up events agents pirel alpharel pre sub subp subc dist)
12  data UpdateM state = Up
13              [state]
14              [Agent]
15              [(BasicPi,state,state)]
16              [(BasicAlpha,state,state)]
17              [(state,Form)]
18              [(state,Prop,Form)]
19              [(state,BasicPi,Pi)]
20              [(state,Agent,Agent)]
21              [state] deriving (Eq,Show)
22
23  {-- Auxiliary Functions --}
24
25  hasPre :: Ord state => Ord state2 => state -> ActPlausM state -> [(state2,Form)] -> state2 -> Bool
26  hasPre w m pre e = case find (\(x,f) -> x == e) pre of
27                      Nothing -> True
28                      Just (x,f) -> isTrueAt m w f
29
30  -- checks whether a given basic pi relation holds between two given states
31  containsBasicPi :: Eq state => [(BasicPi,state,state)] -> BasicPi -> (state,state) -> Bool
32  containsBasicPi pirel p (w,v) = case find (\(p2,x,y) -> (p2 == p) && (x == w) && (y == v)) pirel of
33                      Nothing -> False
34                      Just (_,_,_) -> True
35
36  -- checks whether a given basic alpha relation holds between two given states
37  containsBasicAlpha :: Eq state => [(BasicAlpha,state,state)] -> BasicAlpha -> (state,state) -> Bool
38  containsBasicAlpha alpharel r (w,v) = case find
39                                      (\(r2,x,y) -> (r2 == r) && (x == w) && (y == v)) alpharel of
40                      Nothing -> False
41                      Just (_,_,_) -> True
42
43  -- determines what propositions are true in a state after updating
44  propsTrueInUpdate :: Ord state => Eq state2 => state -> ActPlausM state ->
45                                                          [(state2,Prop,Form)] -> state2 -> [Prop]
46  propsTrueInUpdate w m@(Mo _ _ val _ _ _ _) sub e = case find (\(x,ps) -> x == w) val of
47                      Nothing -> subs
48                      Just (_,ps) -> (wosubs ps) ++ subs
```

```
49                   where
50                       wosubs (x:xs) = case find (\(y,p,_) -> (p == x) && (y == e)) sub of
51                            Nothing -> x:(wosubs xs)
52                            Just (_,_,_) -> (wosubs xs)
53                       wosubs [] = []
54                       subs = [p | (x,p,f) <- sub, x == e, isTrueAt m w f]
55
56 -- determines the control of states after updating
57 newOwners :: Eq state1 => Eq state2 => [(state1,state2)] -> [(state1,Agent)] ->
58                                                 [(state2,Agent,Agent)] -> [((state1,state2),Agent)]
59 newOwners new control sub = [ ((w,e),agent w e) | (w,e) <- new ]
60     where
61     agent w e = case find (\(x,_) -> x == w) control of
62                 Just (x,Ag i) -> case find (\(y,Ag j,_) -> y == e && j == i) sub of
63                      Just (y,Ag j,Ag k) -> Ag k
64                      Nothing -> Ag i
65                 Nothing -> case find (\(y,Ag j,_) -> y == e && j == 0) sub of
66                      Just (y,Ag 0,Ag k) -> Ag k
67                      Nothing -> Ag 0
68
69 {-- Constraint Checking --}
70
71 -- checks whether a given formula is a tautology
72 -- warning: partial implementation
73 isTautology :: Form -> Bool
74 isTautology form | form == Top = True
75                  | otherwise   = False
76
77 -- checks whether a given formula is a falsity
78 -- warning: partial implementation
79 isFalsity :: Form -> Bool
80 isFalsity form | form == Neg (Top) = True
81                | otherwise         = False
82
83 -- checks whether a complex pi program is of a certain (exceptional) form
84 isSafeSubstitution :: Pi -> Bool
85 isSafeSubstitution (PUnion (PConcat (PTest (Prop (C i))) (PConcat (bp1) (PTest (Prop (C j)))) )
86           (PConcat (PTest (Neg (Prop (C k)))) (PConcat (bp2) (PTest (Neg (Prop (C l)))) ) ) )
87                  = and [i == j, j == k, k == l, bp1 == bp2]
```

```
88  isSafeSubstitution _ = False
89
90  -- obtains the inside of the pi program of exceptional form, if the program is of this form
91  -- otherwise does nothing
92  unpackSafeSubstitution :: Pi -> Pi
93  unpackSafeSubstitution p@(PUnion
94                            (PConcat (PTest (Prop (C i))) (PConcat (p1) (PTest (Prop (C j)))) )
95                            (PConcat (PTest (Neg (Prop (C k))))
96                                                          (PConcat (p2) (PTest (Neg (Prop (C l)))) ) ) )
97                   | and [i == j, j == k, k == l, p1 == p2] = p1
98                   | otherwise                              = p
99
100 -- checks whether a given pi program is reflexive on any model
101 -- warning: partial implementation
102 alwaysReflexive :: Pi -> Bool
103 alwaysReflexive (PUnion (PConcat (PTest (Prop (C i))) (PConcat (bp1) (PTest (Prop (C j)))) )
104           (PConcat (PTest (Neg (Prop (C k)))) (PConcat (bp2) (PTest (Neg (Prop (C l)))) ) ) )
105                  = and [i == j, j == k, k == l, bp1 == bp2, alwaysReflexive bp1]
106 alwaysReflexive (PBasic _) = True
107 alwaysReflexive (PConcat pi1 pi2) = (alwaysReflexive pi1) && (alwaysReflexive pi2)
108 alwaysReflexive (PUnion pi1 pi2) = (alwaysReflexive pi1) || (alwaysReflexive pi2)
109 alwaysReflexive (PKleene _) = True
110 alwaysReflexive (PConv pi1) = alwaysReflexive pi1
111 alwaysReflexive (PTest form) = isTautology form
112
113 -- checks whether the equivalence class of a given complex pi program contains the equivalence
114 -- class of a given basic pi program, for any model
115 alwaysContainsEq :: Pi -> BasicPi -> Bool
116 alwaysContainsEq (PBasic (P i)) (P j) = j == i
117 alwaysContainsEq (PConcat pi1 pi2) bp = ((alwaysContainsEq pi1 bp) && (alwaysReflexive pi2)) ||
118                                         ((alwaysContainsEq pi2 bp) && (alwaysReflexive pi1))
119 alwaysContainsEq (PUnion pi1 pi2) bp = (alwaysContainsEq pi1 bp) || (alwaysContainsEq pi2 bp)
120 alwaysContainsEq (PKleene pi1) bp = alwaysContainsEq pi1 bp
121 alwaysContainsEq (PConv pi1) bp = alwaysContainsEq pi1 bp
122 alwaysContainsEq (PTest _) _ = False
123
124 -- obtains the relation for a given basic pi program
125 basicUpdatePi :: Ord event => [(BasicPi,event,event)] -> BasicPi -> Rel event
126 basicUpdatePi rels pi1 = (\x y -> case find
```

```
127                                  (\(pi2,w,z) -> (pi2 == pi1) && (w == x) && (z == y)) rels of
128                         Just _ -> True
129                         Nothing -> False)
130
131  -- obtains the relation for a given complex pi program, in which all tests are replaced by
132  -- the empty relation
133  obtainUpdatePiEta :: Ord event => UpdateM event -> Pi -> Rel event
134  obtainUpdatePiEta u@(Up _ _ pirel _ _ _ _ _ _) (PBasic pi1) = basicUpdatePi pirel pi1
135  obtainUpdatePiEta u@(Up events _ _ _ _ _ _ _ _) (PConcat pi1 pi2) =
136                                  compR events (obtainUpdatePiEta u pi1) (obtainUpdatePiEta u pi2)
137  obtainUpdatePiEta u (PUnion pi1 pi2) = unionR (obtainUpdatePiEta u pi1) (obtainUpdatePiEta u pi2)
138  obtainUpdatePiEta u@(Up events _ _ _ _ _ _ _ _) (PKleene pi1) =
139                                                     transClosure events (obtainUpdatePiEta u pi1)
140  obtainUpdatePiEta u (PConv pi1) = (\x y -> (obtainUpdatePiEta u pi1) y x)
141  obtainUpdatePiEta u (PTest _) = (\x y -> False)
142
143  -- obtains the relation for a given complex pi program, in which all test are replaced by
144  -- the smallest reflexive relation
145  obtainUpdatePiTheta :: Ord event => UpdateM event -> Pi -> Rel event
146  obtainUpdatePiTheta u@(Up _ _ pirel _ _ _ _ _ _) (PBasic pi1) = basicUpdatePi pirel pi1
147  obtainUpdatePiTheta u@(Up events _ _ _ _ _ _ _ _) (PConcat pi1 pi2) =
148                              compR events (obtainUpdatePiTheta u pi1) (obtainUpdatePiTheta u pi2)
149  obtainUpdatePiTheta u (PUnion pi1 pi2) =
150                                   unionR (obtainUpdatePiTheta u pi1) (obtainUpdatePiTheta u pi2)
151  obtainUpdatePiTheta u@(Up events _ _ _ _ _ _ _ _) (PKleene pi1) =
152                                                   transClosure events (obtainUpdatePiTheta u pi1)
153  obtainUpdatePiTheta u (PConv pi1) = (\x y -> (obtainUpdatePiTheta u pi1) y x)
154  obtainUpdatePiTheta u (PTest _) = (\x y -> (x == y))
155
156  -- determines whether control of a given state is changed from one given agent to another given
157  -- agent
158  subCjEqi :: Eq state => [(state,Agent,Agent)] -> state -> Agent -> Agent -> Bool
159  subCjEqi subc e (Ag j) (Ag i) = case find
160                              (\(s,c1,c2) -> (s == e) && (c1 == (Ag j)) && (c2 == (Ag i))) subc of
161                   Nothing -> case find (\(s,c1,c2) -> (s == e) && (c1 == (Ag j))) subc of
162                        Nothing -> j == i
163                        Just _ -> False
164                   Just _ -> True
165
```

```
166 -- determines whether control of a given state is not changed from one given agent to another
167 -- given agent
168 subCjNotEqi :: Eq state => [(state,Agent,Agent)] -> state -> Agent -> Agent -> Bool
169 subCjNotEqi subc e (Ag j) (Ag i) = case find (\(s,c1,c2) -> (s == e) && (c1 == (Ag j))) subc of
170                   Nothing -> case find (\(s,c1,c2) -> (s == e) && (c1 == (Ag j))) subc of
171                        Nothing -> j /= i
172                        Just _ -> False
173                   Just (_,c1,_) -> c1 /= (Ag i)
174
175 -- determines whether there exist a number of states that satisfy a certain condition and of which
176 -- the disjunction of their preconditions constitute a tautology
177 -- warning: partial implementation
178 existCoveringEventsSuchThat :: Eq state => [state] -> [(state,Form)] -> (state -> Bool) -> Bool
179 existCoveringEventsSuchThat events pre f =
180                                    or[exists relevantStates (\s1 -> exists relevantStates (\s2 ->
181                                       case find (\(s,_) -> (s == s1)) pre of
182                                            Nothing -> True
183                                            Just (_,f1) -> case find (\(s,_) -> (s == s2)) pre of
184                                                 Nothing -> True
185                                                 Just (_,f2) -> f1 == (Neg f2)
186                                       )),
187                                       exists relevantStates (\s1 ->
188                                       case find (\(s,_) -> (s == s1)) pre of
189                                            Nothing -> True
190                                            Just (_,f1) -> f1 == Top
191                                       )]
192                   where
193                   relevantStates = [ s | s <- events, (f s) ]
194
195 -- checks whether a given update model does not violate the constraints laid on update models
196 checkUpdateConstraints :: Eq state => Ord state => UpdateM state -> Bool
197 checkUpdateConstraints u@(Up events agents pirel alpharel pre sub subp subc dist) =
198                                                                                 and [c1,c2,c3,c4]
199     where
200     -- the constraint that ensures the product adheres to the constraint of reflexivity
201     c1 = forall events (\e -> forall agents (\(Ag i) ->
202         and[case find (\(s,bp,p) -> (s == e) && (bp == (P i))) subp of
203               Nothing -> (obtainUpdatePiEta u (PBasic (P i))) e e
204               Just (_,_,p) -> obtainUpdatePiEta u (unpackSafeSubstitution p) e e,
```

```
205         case find (\(s,bp,p) -> (s == e) && (bp == (P i))) subp of
206              Nothing -> True
207              Just (_,_,p) -> alwaysReflexive p]
208         ))
209     -- the constraint that ensures the product adheres to the constraint of distinction
210     c2 = forall events (\e1 -> forall events (\e2 -> forall agents (\(Ag i) ->
211          and[exists agents (\(Ag j) -> subCjEqi subc e1 (Ag j) (Ag i)),
212              exists agents (\(Ag k) -> subCjNotEqi subc e2 (Ag k) (Ag i))]
213          `implies`
214          or[case find (\(s,bp,p) -> (s == e1) && (bp == (P i))) subp of
215               Nothing -> not ((obtainUpdatePiTheta u (getEquiv (PBasic (P i)))) e1 e2)
216               Just (_,_,p) -> not ((obtainUpdatePiTheta u (getEquiv p)) e1 e2),
217          case find (\(s,bp,p) -> (s == e1) && (bp == (P i))) subp of
218               Nothing -> False
219               Just (_,_,p) -> isSafeSubstitution p]
220          )))
221     -- the constraint that ensures the product adheres to the constraint of awareness
222     c3 = forall events (\e1 -> forall events (\e2 -> forall agents (\(Ag i) ->
223          forall (differentAlpha alpharel) (\a ->
224          and[((obtainUpdatePiTheta u (getEquiv (PBasic (P i)))) e1 e2),
225          exists agents (\(Ag j) -> subCjEqi subc e1 (Ag j) (Ag i)),
226          exists agents (\(Ag k) -> subCjEqi subc e2 (Ag k) (Ag i)),
227          exists events (\e3 -> case find
228                                 (\(r,s1,s2) -> (r == a) && (s1 == e1) && (s2 == e3))
229                                                                             alpharel of
230                   Nothing -> False
231                   Just _ -> True)]
232          `implies`
233          and[existCoveringEventsSuchThat events pre
234                          (\s -> case find
235                                  (\(r,s1,s2) -> (r == a) && (s1 == e2) && (s2 == s))
236                                                                             alpharel of
237               Nothing -> False
238               Just (_,_,_) -> True),
239          forall agents (\(Ag j) -> (subCjEqi subc e1 (Ag j) (Ag i)) `implies` (j == i)),
240          forall agents (\(Ag k) -> (subCjEqi subc e2 (Ag k) (Ag i)) `implies` (k == i))]
241          ))))
242     -- the constraint that ensures the product adheres to the constraint of nondeterminism
243     c4 = forall events (\e1 -> forall events (\e2 -> forall events (\e3 ->
```

```
244           forall agents (\(Ag i) -> forall (differentAlpha alpharel) (\a ->
245           and[exists agents (\(Ag j) -> subCjEqi subc e1 (Ag j) (Ag i)),
246           case find (\(r,s1,s2) -> (r == a) && (s1 == e1) && (s2 == e2)) alpharel of
247                     Nothing -> False
248                     Just _ -> True,
249           case find (\(r,s1,s2) -> (r == a) && (s1 == e1) && (s2 == e3)) alpharel of
250                     Nothing -> False
251                     Just _ -> True
252           ]
253           `implies`
254           let y = case find (\(s,bp,p) -> (s == e2) && (bp == (P i))) subp of
255                 Nothing       -> (PBasic (P i))
256                 Just (_,_,p) -> unpackSafeSubstitution p
257              in
258           and[obtainUpdatePiEta u (getEquiv y) e2 e3,
259           alwaysContainsEq y (P i),
260           forall agents (\(Ag j) -> (subCjEqi subc e2 (Ag j) (Ag i)) `implies` (j == i))]
261           )))))
262
263 {-- Update Functions --}
264
265 -- obtains the update product
266 update :: Ord state1 => Ord state2 => UpdateM state2 -> ActPlausM state1 ->
267                                                                  ActPlausM (state1,state2)
268 update u@(Up _ _ _ _ _ _ subp _ _) m =
269        let m1@(Mo states agents val pirel alpharel control dist) = supdate u m in
270        Mo states
271           agents
272           val
273           [ (P x,w,v) | x <- map (\(Ag x) -> x) agents, w@(w1,w2) <- states, v <- states,
274                                   case find (\(e,(P y),_) -> (x == y) && (e == w2)) subp of
275                Nothing -> elem (P x,w,v) pirel
276                Just (_,_,rel) -> inR (obtainPi m1 rel) (w,v) ]
277           alpharel
278           control
279           dist
280
281 -- obtains the partial update product, in which no substitution on pi programs has taken place
282 supdate :: Ord state1 => Ord state2 => UpdateM state2 -> ActPlausM state1 ->
```

```
283                                                                 ActPlausM (state1,state2)
284 supdate u@(Up events agents2 pirel2 alpharel2 pre sub subp subc dist2)
285         m@(Mo states agents1 val pirel alpharel control dist1)
286         = let newStates = [(x,y) | x <- states, y <- events, hasPre x m pre y] in
287           Mo newStates
288              [ a | a <- agents1, elem a agents2 ]
289              [ ((x,y),(propsTrueInUpdate x m sub y)) | (x,y) <- newStates ]
290              [ (p,(w,e),(v,f)) | (w,e) <- newStates, (v,f) <- newStates, (p,x,y) <- pirel,
291                            x == w, y == v, ((containsBasicPi pirel2 p (e,f)) || (e == f))]
292              [ (r,(w,e),(v,f)) | (w,e) <- newStates, (v,f) <- newStates, (r,x,y) <- alpharel,
293                                      x == w, y == v, containsBasicAlpha alpharel2 r (e,f)]
294              [ (w,a) | (w,a) <- newOwners newStates control subc, a /= (Ag 0) ]
295              [ (x,y) | x <- dist1, y <- dist2, elem (x,y) newStates]
```

A.5 Display

The module *Display* contains functions that display models using Graphviz.

```
1  module Display
2
3  where
4
5  import System
6  import List
7
8  import SetOrd
9  import Rel
10 import ModelChecker
11 import Updates
12
13 {-- Displaying Models Using GraphViz --}
14
15 showModel :: Eq a => Show a => ActPlausM a -> IO ()
16 showModel m = do writeFile "model.dot" (writeP m)
17                  system "dot -Tpng model.dot -o model.png"
18                  system "open model.png"
```

```
19                   return ()
20
21  showUpdateModel :: Eq a => Show a => UpdateM a -> IO ()
22  showUpdateModel m = do writeFile "updatemodel.dot" (writeUpdateP m)
23                         system "dot -Tpng updatemodel.dot -o updatemodel.png"
24                         system "open updatemodel.png"
25                         return ()
26
27  {-- Auxiliary Functions --}
28
29  -- replaces one character by another in a string
30  replace :: Char -> Char -> String -> String
31  replace a b (x:xs) | x == a     = b:(replace a b xs)
32                     | otherwise = x:(replace a b xs)
33  replace _ _ [] = []
34
35  -- removes a given character from a string
36  remove :: Char -> String -> String
37  remove a (x:xs) | x == a     = remove a xs
38                  | otherwise = x:(remove a xs)
39  remove _ [] = []
40
41  -- constants used internally for GraphViz formatting
42  c1 :: Char
43  c1 = 'l'
44  c2 :: Char
45  c2 = 'r'
46  c3 :: Char
47  c3 = 'c'
48  c4 :: Char
49  c4 = 'q'
50
51  {-- Display Functions --}
52
53  -- writes down all information about a model in GraphViz format
54  writeP :: Eq a => Show a => ActPlausM a -> String
55  writeP (Mo states agents val relpi relalpha control dist) = "digraph M {\n" ++
56      writeStates states ++
57      writePi sortPi ++
```

```
58      writeAlpha relalpha ++
59      "textbox [shape=box, label=\"" ++
60      writeVal (filterVal sortVal) ++
61      writeC control ++
62      "\"];}\n"
63          where
64          sortPi = sortBy (\(i,w,v) (j,w2,v2) -> compare i j) relpi
65          -- writes down information about states
66          writeStates [] = ""
67          writeStates (x:xs) = 'w':(replace '"' c4 (replace ',' c3 (replace ')' c2 (replace '('
68                  c1 (show x))))) ++ " [shape=" ++ (maybeDouble x) ++ "circle, label=\"" ++
69                  (remove '"' (show x)) ++ "\"];\n" ++ writeStates xs
70          maybeDouble x | elem x dist = "double"
71                        | otherwise   = ""
72          -- writes down information about pi programs
73          writePi [] = ""
74          writePi (x@(P i, w, v):xs) = 'w':(replace '"' c4 (replace ',' c3 (replace ')' c2
75                  (replace '(' c1 (show w))))) ++ " -> w" ++ (replace '"' c4 (replace ',' c3
76                  (replace ')' c2 (replace '(' c1 (show v))))) ++ " [style=dotted, label=\"p" ++
77                  show i ++ (printIndices xs) ++
78                  "\"];\n" ++ writePi (filter (not.sameArrow) xs)
79              where printIndices [] = []
80                    printIndices (y@(P j, _, _):ys) | sameArrow y = ',':show j ++ printIndices ys
81                                                    | otherwise   = printIndices ys
82                    sameArrow (P j, w2, v2) = (w == w2) && (v == v2)
83          -- writes down information about alpha programs
84          writeAlpha [] = ""
85          writeAlpha (x@(R i, w, v):xs) = 'w':(replace '"' c4 (replace ',' c3 (replace ')' c2
86                  (replace '(' c1 (show w))))) ++ " -> w" ++ (replace '"' c4 (replace ',' c3
87                  (replace ')' c2 (replace '(' c1 (show v))))) ++ " [style=solid, label=\"a" ++
88                  show i ++ (printIndices xs) ++ "\"];\n" ++ writeAlpha (filter (not.sameArrow) xs)
89              where printIndices [] = []
90                    printIndices (y@(R j, _, _):ys) | sameArrow y = ',':show j ++ printIndices ys
91                                                    | otherwise   = printIndices ys
92                    sameArrow (R j, w2, v2) = (w == w2) && (v == v2)
93          -- filters recoverable (counter) propositions from the valuation
94          filterVal v = map (\(w,ps) -> (w,(filterValProp ps))) v
95          filterValProp (p1@(V i j):(p2:ps)) | p2 == (V i (j+1)) = filterValProp (p2:ps)
96                                             | otherwise         = p1:(filterValProp (p2:ps))
```

```
97          filterValProp (p1:(p2:ps)) = p1:(filterValProp (p2:ps))
98          filterValProp (p:[]) = [p]
99          filterValProp [] = []
100         sortVal = map (\(w,ps) -> (w,(sortBy valSort ps))) val
101         valSort (V i1 j1) (V i2 j2) | i1 == i2  = compare j1 j2
102                                     | otherwise = compare i1 i2
103         valSort (V _ _) _ = GT
104         valSort _ (V _ _) = LT
105         valSort _ _ = EQ
106         -- writes down information about the valuation
107         writeVal [] = ""
108         writeVal (x@(s,ps):xs) = "V(" ++ (remove '"' (show s)) ++ ")=" ++ show ps ++ "\\n"
109                                  ++ writeVal xs
110         -- writes down information about the control function
111         writeC [] = ""
112         writeC (x@(s,a):xs) = "C(" ++ (remove '"' (show s)) ++ ")=" ++ show a ++ "\\n"
113                               ++ writeC xs
114 
115 -- writes down all information about an update model in GraphViz format
116 writeUpdateP :: Eq a => Show a => UpdateM a -> String
117 writeUpdateP (Up events agents relpi relalpha pre sub subp subc dist) = "digraph M {\n" ++
118     writeStates events ++
119     writePi sortPi ++
120     writeAlpha relalpha ++
121     "textbox [shape=box, label=\"" ++
122     writePre pre ++
123     writeSub sub ++
124     writeSubP subp ++
125     writeSubC subc ++
126     "\"];}\n"
127         where
128         sortPi = sortBy (\(i,w,v) (j,w2,v2) -> compare i j) relpi
129         -- writes down information about states
130         writeStates [] = ""
131         writeStates (x:xs) = 'w':(replace '"' c4 (replace ',' c3 (replace ')' c2 (replace '('
132                 c1 (show x))))) ++ " [shape=" ++ (maybeDouble x) ++ "circle, label=\"" ++
133                 (remove '"' (show x)) ++ "\"];\n" ++ writeStates xs
134         maybeDouble x | elem x dist = "double"
135                       | otherwise   = ""
```

```haskell
136   -- writes down information about pi programs
137   writePi [] = ""
138   writePi (x@(P i, w, v):xs) = 'w':(replace '"' c4 (replace ',' c3 (replace ')' c2
139           (replace '(' c1 (show w))))) ++ " -> w" ++ (replace '"' c4 (replace ',' c3
140           (replace ')' c2 (replace '(' c1 (show v))))) ++ " [style=dotted, label=\"p" ++
141           show i ++ (printIndices xs) ++ "\"];\n" ++ writePi (filter (not.sameArrow) xs)
142       where printIndices [] = []
143             printIndices (y@(P j, _, _):ys) | sameArrow y = ',':show j ++ printIndices ys
144                                             | otherwise   = printIndices ys
145             sameArrow (P j, w2, v2) = (w == w2) && (v == v2)
146   -- writes down information about alpha programs
147   writeAlpha [] = ""
148   writeAlpha (x@(R i, w, v):xs) = 'w':(replace '"' c4 (replace ',' c3 (replace ')' c2
149           (replace '(' c1 (show w))))) ++ " -> w" ++ (replace '"' c4 (replace ',' c3
150           (replace ')' c2 (replace '(' c1 (show v))))) ++ " [style=solid, label=\"a" ++
151           show i ++ (printIndices xs) ++ "\"];\n" ++ writeAlpha (filter (not.sameArrow) xs)
152       where printIndices [] = []
153             printIndices (y@(R j, _, _):ys) | sameArrow y = ',':show j ++ printIndices ys
154                                             | otherwise   = printIndices ys
155             sameArrow (R j, w2, v2) = (w == w2) && (v == v2)
156   -- writes down information about preconditions
157   writePre [] = ""
158   writePre (x@(s,f):xs) = "pre(" ++ (remove '"' (show s)) ++ ")=" ++ show f ++ "\\n"
159                           ++ writePre xs
160   -- writes down information about the substitution of propositions
161   writeSub [] = ""
162   writeSub (x@(s,p,f):xs) = "sub(" ++ (remove '"' (show s)) ++ ")(" ++ show p ++ ")="
163                             ++ show f ++ "\\n" ++ writeSub xs
164   -- writes down information about the substitution of pi programs
165   writeSubP [] = ""
166   writeSubP (x@(s,bp,p):xs) = "subp(" ++ (remove '"' (show s)) ++ ")(" ++ show bp ++ ")="
167                               ++ show p ++ "\\n" ++ writeSubP xs
168   -- writes down information about the substitution of control
169   writeSubC [] = ""
170   writeSubC (x@(s,i,j):xs) = "subc(" ++ (remove '"' (show s)) ++ ")(" ++ show i ++ ")="
171                              ++ show j ++ "\\n" ++ writeSubC xs
```