# Modeling belief revision in multi-agent systems

## *Design of a dynamic epistemic logic tool to reason about belief change on plausibility models*

Gijs Hofstee
March 2012

**Master Thesis**
Artificial Intelligence
University of Groningen, The Netherlands

Internal supervisor:
Dr. Sonja Smets (Artificial Intelligence, University of Groningen)

Second supervisor:
Prof. dr. Rineke (L.C.) Verbrugge (Artificial Intelligence, University of Groningen)

university of groningen / faculty of mathematics and natural sciences / artificial intelligence

# Abstract

With the advance of robots and more intelligent computer programs, belief revision is becoming an increasingly important field of study as it allows agents to revise their views of the world using a logic as a basis for the model. Various areas of science, like robotics, mathematics and philosophy however design their own solutions for belief revision. In the first part of this paper, have compared three different ways to deal with belief revision, and determined if and how much overlap they show. As it turns out, for a combination of private and public updates and upgrades, Dynamic Epistemic Logic (DEL) is the most logical choice. In the second part of this paper, we explain the essential parts of a program we have designed to evaluate any logical formulas on a DEL plausibility model. In addition, some design choices are highlighted, so anyone interested could reproduce a similar program. The main purpose of this program is to show that such updates can be done in an automated fashion (this has not been done before). It also results in a viable benchmark or starting point for future work in this area. Finally, we shall show how this program works, as well as prove it does actually work by using it to solve the muddy children puzzle.

# Index

# Chapter 1 - Introduction

We all use logic every day, though not in the mathematical way it is often taught to students in their first year. The logic we use in daily life is not just about truth and falsity or about absolute certainty. Instead, our reasoning is often based on less than certainty; on what is possibly true or probably true. If someone tells you something, then before accepting what he tells you, you first evaluate how likely it is that their information might actually be true. When someone is an expert on a subject, you might be convinced and change your mind, even if their information contradicts your own beliefs. If they are not an expert however, you might disregard their information and stick to your own beliefs. This shows that besides the static true/false world, there is the world of plausibility and belief revision. In a multi-agent context, it becomes even more complicated when agents start to reason about the beliefs of others. If it is our aim to design intelligent and rational artificial agents, it is crucial that we fully understand and can model complicated belief revision processes. Designing a rational belief revision strategy is an aspect of intelligent behavior. While research on non-monotonic reasoning is an essential part of AI, how to model higher-order belief revision scenarios has received less attention. This is not very surprising if we observe that the field of belief revision in logics for multi-agent systems has only been recently developed. As such, there are still parts missing, like how to validate plausibility models and how to perform updates on them. These ingredients are essential to make a computer work with belief revisions, and will be part of the research in this thesis.

## *1.1- Research question*

There are many ways in which belief revision can be modeled, some methods are more quantitative and others are more qualitative. We make a selection in this thesis of three such methods that we will analyze in the first, theoretical part. In the final part of this thesis, one of these three methods will be chosen in order to implement our practical tool to model belief revision. Requirements for this tool will be that it can deal with higher order believes (I believe that you believe, etc), as well as public and private updates and upgrades.
To this end, the research questions are split in a theoretical part and a practical part, just as the rest of this thesis.

**Theory based questions**:

- What are the differences between Dynamic Epistemic Logic (DEL) based plausibility models, Jeffrey conditioning and weak interpreted systems?
- Additionally, which pro's and con's are associated with each of these techniques?

To answer the second question we shall look at three scenario's. The first is when agents are not certain about new incoming information, but every communication is public and truthful. The second will be when not all

information is public, private announcements can be made by agents to "cheat", but all information is still truthful. (Baltag, 2002 contains a good explanation of this problem, and offers an initial solution to it, though his modeling tools in 2002 are not based on plausibility models.) The third scenario is when we allow both private information and false information; agents are then deliberately lying to mislead other agents.

**Practical Questions**:
For the practical questions, we will use DEL based plausibility models. This is partially because we expect that both interpreted systems and Jeffrey conditioning will run into problems with higher order beliefs (I believe that you believe, etc), and in part due to the success of the DEMO model checker (Van Eijk, 2004).
The practical questions we look at are the following:

- How to check plausibility models? Checking the validity of a DEL based plausibility model has not been done before, but it forms an essential link to actually working with these models.
- How to model and implement dynamic updates on a plausibility model? Solving this question requires plausibility model checking. Making dynamic updates on a plausibility model is the essence of this research. As a case study we will consider the muddy children scenario adapted to a belief context. In this scenario, the children will typically not consider the source of information (the statement of the father in the standard scenario) to be infallible and hence, this will affect the way they revise their beliefs. The solution to this will allow us to perform dynamic belief revision in the context of Dynamic Epistemic Logic, and thus provide a benchmark for future comparative studies.
- How to implement answers to the previous two questions in software? Once we know the rules, we do of course want to make a program that can apply the rules for us (and our fellow researchers), to allow for easy model checking, and to serve as a benchmark for future work.

# Chapter 2 - Theoretical Background

Much research has already been done on Dynamic Epistemic Logic (DEL), as can been seen in Van Ditmarsch et al. (2007). This includes both the design of models and the tools for model checking. DEL can be used to model knowledge and belief in a dynamic environment. Because we actually will focus on model checking, we refer to Halpern & Vardi, 1991.

For reasoning about uncertainty (from plausibility to probability), there are various approaches. On the one hand there are DEL based plausibility models, as can be seen in Baltag & Smets (2008). This approach uses plausibility models to keep track of which situations an agent considers more plausible. This information is then used to evaluate statements. New information will lead to updates of the plausibility models, depending on how strong the belief in the source of this new information is. Also see (Baltag, Ditmarsch & Moss, 2008). On the other hand there are approaches based on Jeffrey conditioning (Jeffrey, 1992), which is a variation of Bayes' Theorem. And finally, J. Halpern and his co-authors have also proposed an "interpreted systems" approach which has been adapted in the literature to model belief revision with unreliable observations as in Boutilier et al. (1998). These three approaches (DEL based plausibility models, Jeffrey Conditioning and Interpreted Systems) are important paradigms in the current field of belief revision and reasoning with uncertain information. Each of these approaches has its own set of merits and disadvantages, which we will further investigate as part of this research.

## *2.1 Jeffrey conditioning*

Before we can take a look at Jeffrey Conditioning, we need to take a look at Bayes' Theorem (or Bayesian Probability) which is the underlying principle (Bayes, 1764). The definition of this theorem is as follows:

$P(A|B) = (P(B|A)\, P(A)\, ) / P(B)$

Where:

- P(A) is the prior probability of A. By prior we mean that it ignores B, it does however not say anything about the order of events A and B.
- P(B) is the prior probability of B
- P(A|B) is the conditional probability of A, given B. Or the posterior probability of A because it depends upon the value of B.
- P(B|A) is the conditional probability of B, given A.

Suppose there is a school with 60% boys and 40% girls as its students. The female students wear trousers or skirts in equal numbers; the boys all wear trousers. An observer sees a (random) student from a distance, and what the observer can see is that this student is wearing trousers. What is the probability this student is a girl? The correct answer can be computed using Bayes' theorem.

The event *A* is that the student observed is a girl, and the event *B* is that the student observed is wearing trousers. To compute P(*A*|*B*), we first need to know:

- P(*B*|*A*), or the probability of the student wearing trousers given that the student is a girl. Since girls are as likely to wear skirts as trousers, this is 0.5.
- P(*A*), or the probability that the student is a girl regardless of any other information. Since the observer sees a random student, meaning that all students have the same probability of being observed, and the fraction of girls among the students is 40%, this probability equals 0.4.
- P(*B*), or the probability of a (randomly selected) student wearing trousers regardless of any other information. Since half of the girls and all of the boys are wearing trousers, this is 0.5×0.4 + 1.0×0.6 = 0.8.

Given all this information, the probability of the observer having spotted a girl given that the observed student is wearing trousers can be computed by substituting these values in the formula:

(0.5 * 0.4) / 0.8 = 0.25 = P(A|B)

Using Bayes' theorem, one can thus calculate the probability of any such event. If we continue that line of thought, we can also apply it to logic. For example, we can assign various probabilities to states, to determine which is more likely (in an uncertain system).
Now this is very useful for a static situation, but not for a dynamic situation, where the system changes from one state to another given external factors. This is where Jeffrey conditioning (Jeffrey, 1992) comes in. It provides a means to change the probability of a certain state by updating the probability with an event.

Jeffrey's rule is the following:
P2(A) = ( P1(A|B) P2(B) ) + ( P1(A|~B) P2(~B) )

Where P1 denotes the probability before an update, and P2 the probability after. Also "~" here means "not".
So suppose there is an event, with a chance of being canceled when it rains of 0.6, and when it doesn't rain of 0.1, so if we learn it rains, the chance P2(A) becomes 0.6 * 1 + 0 = 0.6.
Now if we learn that the chance of rain is 0.8, we get P2(A) = 0.6 * 0.8 + 0.1 * 0.2 = 0.5 as the overall chance of the event being cancelled.
The first thing we notice is that the chance for P2(A|B) is identical to P(A|B), so that chance does not change at all. Instead, upon learning new data P2(B), only the chance for our event P2(A) gets changed. This is not always something we want, for example, if there is a chance for placing a plant in a room P(B), and a chance for a big plant P(A), then learning that there is sunlight in the room, will lead to an increase of both P(A|B) and P(A). After all, we will be more likely to

put it in a room if we know there is sunshine there. (Zhao & Osherson, 2010) For most cases in logic however, we can safely ignore this, so we will not look at this any further.

We can also include past experience (or knowledge) in this equation, namely P(A|B, f), where f is our past experience (or knowledge) factor, which can also influence the likeliness of some event. But if our aim is to do belief revision, then not our previous knowledge, but rather our amount of trust in the source of this new information should dictate if we belief it or not. Another potential problem with Jeffreys rule is that there does not appear to be any way to reject either Falsum or otherwise bad information. In other words, all information we learn is treated equally, while it most certainly is not always the case in any real situation. We would for example want to trust information from our high school teacher over contradictory information from a stranger. Should this stranger turn out to be a university teacher however, we may want to trust him more. In neither case do we want to trust them equally as Jeffrey conditioning does.

Benferhat (Benferhat et al, 2009) make a framework that adapts Jeffreys rule to possibility theory. By doing this, they can use worlds containing a set of facts to model various possible situations. These worlds are then ordered awarding them a value based on their probability compared to the real world. This way, they show most forms of belief revision are possible. However, they do also mention that Jeffrey conditioning for belief revision updates results in all worlds with P(x) = 1 being “fully plausible”, which of course all consistent epistemic states should be. This will however lead to large amounts of worlds, which are all fully plausible, but have no ordering whatsoever. Again, this is not very desirable, since we usually do have a preference for a certain kind of worlds. For example, if we consider that there is no real proof the Easter bunny does not exist, someone (a child for example) would prefer any worlds where the Easter bunny does actually exist. Even though there are no events that suggest there should be a difference between worlds where he does or does not exist. This means using Jeffrey conditioning in this way, we cannot state preferences (or conditional beliefs), and are thus stuck with a very large amount of possible worlds in which we do not really have any belief preference.

In conclusion, if we try to map the Bayes Theorem and Jeffrey Conditioning back to belief revision logic, it would seem (simplified) that Bayes is able to make updates (take all new data, throw away everything that is not compatible with it), while Jeffrey’s rule makes upgrades (award higher likeliness to new data, but keep old data as “possible”, no matter how low the probability becomes). It is however not possible for Jeffrey Conditioning to do a “weak update”, where only the most likely world with the new data gets promoted (and the rest of the worlds matching that new data remain in their old order) which is needed if new information is "barely" believed, that is to say, if cheating is considered possible by the agent, but not likely, nor is it possible to show preferences or conditional beliefs. Finally, there is also no (obvious) way to have different levels of trust or

reliability in incoming data, or possibilities to deal with potentially incomplete or incorrect data (cheating and lying).
Thus, while Jeffrey conditioning can certainly be used for some purposes in belief revision logic, it cannot (at least not without any serious modification) do everything we would likely to be able to do with it, namely it cannot be used to model scenario 2 and 3 from our research questions.

## *2.2 Interpreted Systems*

Interpreted systems in this article will refer to the system described in a paper by Friedman and Halpern (Friedman & Halpern, 1997) and in particular, belief revision and updates, as elaborated in its companion paper (Friedman & Halpern, 1999).
An interpreted system is made up by the following elements. An agent has a local state, which consists of a set of pairs of (observation, time). These pairs consist of all possible observations an agent has made at any time. The local states of all agents, together with the actual "real world" (usually referred to as environment), make up the set of global states, which are a Cartesian product of the possible local states for each agent. A single global state (or world) is thus the combination of the local states of all agents at one point in time. The set of all global states is also known as a run, and can possibly be infinite. A system, in turn is made up of a set of runs which encompass all possible sequences of events that could occur. Additionally, time is assumed to be expressed in natural numbers usually marking seconds, or the time it takes to make an observation. Finally, the agent is assumed to have perfect recall and is synchronous, which is to say that all states are observed in exact order, with nothing between observations, so there is "nothing" between observation 1 and observation 2, and observation 1 and 2 are encountered exactly in that (serial) order (there are no parallel observations). An interpreted system is such a system together with a mapping PI that associates a truth assignment with the primitive propositions at each state of the system. An interpreted plausibility system is like a Kripke structure for knowledge. Two points of time in certain runs are said to be indistinguishable to an agent when the two local states for that agent are identical. This is consistent with the notion that a local state contains all information the agent has at a certain point in time. When we take all indistinguishable points in a run for a specific agent, we get what in a Kripke structure would be defined as a K relation. For belief, each point (r, m), with r being a run and m being a time index, has a plausibility assignment associated with it, and for each of these plausibility assignments the agent can determine if a point(r,m) is more, less or equally plausible than another.
While the details of how plausibility is determined are originally left open, the authors suggest that some relation between time n and n+1 would be desired. A proposed technique for this is suggested in the form of Bayesian probabilities, using priors on the runs of systems. This approach is analogues to what (Halpern & Tuttle, 1993) does when defining how an agents' probability distribution is changed in a multi-agent system. An agent then uses normal conditioning (as commonly used in probability theory) to change the priors when new information is learned. In this particular case, the authors suggest to condition

assessment made at each point on all runs which are still considered possible at that point. Another important point to note here, is that the authors consider only true input. That means, only true facts can be observed by the agent, and believes are only used to differentiate between states which contain all facts observed so far. For example, if an agent determines that either A or B or both is the case, it can show preference for either A or B over the combination of the two. It will however not consider that the input itself may be faulty. However, for our purposes we are interested in a system that could not only be used with incomplete information, it should ideally also be able to deal with false information.

Another thing is that it is not hard to see that a system containing all possible runs takes enormous (possibly infinite) resources. So to make this workable, a reasonable limit would be that any agent is only aware of all observations it has made so far, thus ignoring all other runs it "could" have made, as well as any possible observations it has not made yet. This does of course introduces other problems, like what to base the plausibility assignments on? In another paper (Boutilier, Friedman & Halpern, 1998) a variation to this framework is considered, with a few more practical considerations in mind.

For example, even with the above limitations in mind, as soon as an agent is running for a long time, or the world is very complex, the sheer amount of past observations and calculations required to determine the current believe at each step will outgrow the storage and calculating power of a regular computer. To reduce the workload and storage, they suggest to use a Markov chain (Markov, 1971) to compute a plausibility value of each local state. Doing this only requires one to have the last state to determine how likely the observation is, and thus determine how it should affect the believes. This usually requires less computation than the method suggested by (Friedman & Halpern, 1997).
They also point out that their version of this framework is generally compatible with the AGM theory (Alchourrón, Gärdenfors, & Makinson, 1985). The part of AGM that belief revision with unreliable observations mostly takes offense to, is the success postulate, which after learning a certain statement Phi, it must also be accepted in the agents belief set (the agent must believe it). While if you have unreliable observations (or cheating or lying agents), this rule seems not very reasonable. After all, if an agent observes 200 times A and then one time not A, it is likely a glitch in the sensor, so we would want to ignore this observation. Only if we observe A multiple times do we want to revise our belief. In their paper, a few restrictions are described which allow an observation system using Markov chains to save this postulate, but the authors note that these restrictions are not applicable to all naturally occurring observation systems.
The AGM postulates also assume the world is static, this requirement is met by the environment state, which never changes. This does lead to some practical issues however, since normally an agent which acts over time does affect the environment in some way. And even if it does not affect the environment itself, some other agents may have an effect on it. In other words, if there is a tree next to the agent, and this is recorded in the environment state, this tree can't fall over

in a storm or because the agent hits it with an axe. By itself this does not exclude belief revision, an agent can still change its believes about the tree, but for practical purposes, a static environment is a drawback which runs against the idea of making new observations over time (why would one if there is nothing new to observe?).
Finally, this implementation of the framework uses Spohn's ordinal conditional functions for k-ranking (Goldszmidt & Pearl, 1996) and (Spohn, 1987) as a form of epistemic entrenchment, since AGM leaves this point open.
Now when an observation is made, a new model needs to be created, this is done by Spohn's conditioning operation (Spohn, 1987). This means effectively that Bayes rule is used to combine prior rankings and new evidence (the actual observation). Unlike Jeffreys rule, this operation does not question if our observation was actually made, but instead it questions if it is a truthful observation. Thus, it question if the source is a good source, rather than if what the source has to say is true. The only problem with it is that in the Interpreted Systems model, the "trust" in the truthfulness of an observation is based on previous observations, rather than on real "trust" in the actual source, which is what we would prefer for the purposes of this paper.

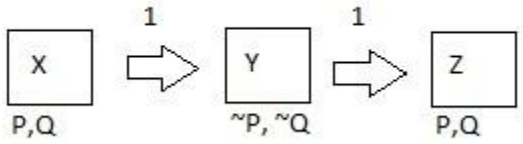
Concluding, we can say that certain forms of Interpreted Systems are (largely) compatible with existing AGM theory, and are also largely applicable to practical problems. Both belief revision and updates are possible with Interpreted Systems in general. Belief revision with incomplete information is also possible. When there are agents or sensors providing false information (lying), some forms of Interpreted Systems provide a solution, but it may in some cases break with standard AGM theory. Additionally, it is not possible to assign different levels of trust to different sources of information. The normal action of this type of system is conservative upgrades, radical upgrades on the other hand do not appear to be possible without changing the model (to include levels of trust in certain sources). Also public truthful announcements are a problem when dealing with a version of Interpreted Systems suited for unreliable observations.
So while Interpreted Systems provide a different set of possibilities than Jeffrey Conditioning, it still does not fulfill all the requirements (which is depending on which implementation or adaptation of Interpreted systems is used) we made in the initial research question (namely public truthful announcements without private announcements, a combination of public and private announcements, and finally announcements of false information).

## *2.3 DEL based plausibility models*

Dynamic Epistemic Logic (DEL) forms the basis of this technique. So we shall provide a short summary of what DEL is. Various papers describe DEL, and what it is based on. This section in particular uses (Baltag, Ditmarsch & Moss, 2008).

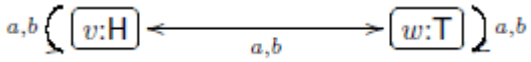
In DEL, each world has a valuation that determines which logical propositions are true and which are false (on which basic logical operators like AND, OR, NOT, IMPLIES and EQUALS can be used) , in turn, a model contains of any

amount of such worlds making up all possible worlds in the model. A model also contains one or more agents, who have a certain set of beliefs and knowledge, this is modeled by adding relations between worlds. For knowledge, an equivalence relation is commonly used, while for belief a plausibility relation is used. It should however be noted that when there is an equivalence relation between two worlds, they must also have a plausibility relation (to determine which, if any is preferred, without it, the sentence "knowing P implies believing P" would not hold). Additionally, a plausibility relation can only exist between two worlds which are indistinguishable to the agent in terms of knowledge. This means that when drawing the model, it suffices to only draw plausibility relations, since one knows that there must also be equivalence relations. All relation have a few similarities, namely that each belongs to one specific agent, and that a relation goes from one world to another (except reflexive relations which come and go to the same world). In terms of relations, each world is reflexive for each agent, and all worlds which are connected to each other by relations for a certain agent are transitive and connected. Equivalence relations are also symmetric, which combined with reflexivity and transitivity makes up the equivalence property, hence the name. Reflexive, transitive and symmetric are well defined in literature, in this case we mean by connected that a world, or a group of worlds can always be compared to each connected world or group of worlds and be found either equally plausible or more plausible.
Belief in this case can be defined in terms of relations. The world (or group of worlds) with the most relations going to it in a group of connected worlds is considered the most plausible. Thus, the belief of an agent on a certain world X is defined as the world with the most incoming relations which can be reached from world X by relations of our agent.
A simple example with 3 worlds (X, Y, Z):



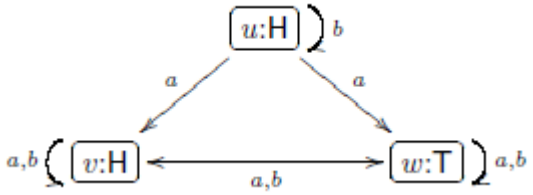

In this example, both reflexive and transitive relations have been omitted, but it can be easily seen that world Z has the most incoming relations for agent 1 (one reflexive, one transitive from X, and one directly from Y). Using this, we could ask if agent 1 believes on world Y that the formula "P AND Q" holds. In this case, the agents belief is world Z, on which the evaluation takes place. So while "P AND Q" is not true on world Y itself, it is on world Z, and thus it is the agents belief that "P AND Q" is true.
Now next to belief, there is also knowledge. Knowledge is defined in DEL as any formula which holds true in all worlds an agent considers possible (all worlds connected by relations) as seen from a certain world. So in our example that would require a statement to be true in all three our worlds (X, Y and Z). However, if we added a fourth world, called W and do not add any relations except the reflexive one, then on that world knowledge would constitute just any statement true in W (and likewise, on world X, Y and Z, knowledge would still be

any statement true on all three worlds, as W does not have any relations to other worlds for this agent).
This grouping of worlds which are connected by relations of a single agent is called an equivalence class. Within one equivalence class, an agent has the same belief in all worlds, namely, those true in the most plausible world of that equivalence class. This also allows to make statements about strong and weak belief. Some formula is considered a strong belief if all worlds in which the formula are true are strictly more plausible than any worlds in which the formula is false. Weak belief is any formula which is true in the most plausible world, but there are also worlds in which the formula is true which are less plausible than worlds in which it is false. To take this back to our example, if a formula is true in world Z, or Z and Y, but not in X, it would be a strong belief, but if it was true in Z and X, but not Y, then it would be a weak belief (as "P AND Q" is in our example).
Using this, we can of course also define updates as well as radical and conservative upgrades. An update or upgrade is an event (or action) observed by the agent(s), and changes an existing model into another one. An upgrade or update, being an event, always describes some statement (a logical formula phi) which was observed. ## After a radical upgrade with a certain logical formula on an initial model, that formula would become a strong belief, while after a conservative upgrade, that formula would become only weakly believed after the upgrade action. Likewise, an update, for one agent, is simply removing relations going from or to the world(s) in which the update statement is true (while retaining the relations amongst the worlds which are true according to the update).
Radical and conservative upgrades, along with updates are what makes up the "dynamic" part of DEL, but the above text only mentions what would happen for a single agent case (for how agents fit in the model, see the beginning of chapter 2.3). For a multi agent case, if the information were a public announcement, the changes would be identical, with just the degree of faith in the source determining if an update is to be used, or a radical upgrade, or a conservative upgrade. It gets more complex when private announcements are also added.
Consider a coin, which has two sides (heads and tails, or H and T), and two agents, A and B. If they both don't know which side is up, the scenario can be drawn as follows:




However, if agent B knows which is which, as he learned it by taking a peak at the coin or because someone told him (which can be considered a private

announcement). Agent A does still not know the state of the coin, but also does not know about agent B's knowledge. This leads to the following model:




As can be seen, in world U, the real world, agent B knows the answer, but when asked what she believes, agent A would believe the world(s) with the most relations (for agent a) going to them, in this case, V and W. She would also believe agent B does not know the answer either. We observe here, that for private announcements a model will contain more (copies of) worlds, as well as more relations. And alternately, when using public announcements, the model contains less (or equal) amounts of relations. For full details on dynamic belief revision, we refer to (Baltag & Smets, 2008). Additionally, a more practical minded approach of how one can create a new state after an update will be given in chapter 3.

Another issue to consider is the difference between static worlds, and changing worlds. In a static world, the reality does not change, and we can use conditional beliefs to talk about preferences. In a changing world, this does not hold, instead we use updates or upgrades, and when we talk about higher order belief (agent A believes that agent B...) any update or upgrade causes a shift in higher order belief as well, thus effectively changing the world (and not just one agents knowledge of it), even if the physical world never changed. Changing higher order belief however introduces another oddity, namely Moore sentences (Moore, 1912 ). Moore sentences actually change the epistemic situation when they are used as updates or upgrades (updating a model with "P AND NOT belief P" for example). But even more important, when Moore sentences are used in updates or upgrades, they will be false after the update or upgrade has been performed. Unlike standard AGM (Alchourrón, Gärdenfors, & Makinson, 1985) or AGM based solutions, DEL based models can deal with Moore sentences in updates and upgrades, since they drop the requirement for a success axiom of AGM (after being updated with p, the agent must believe P). The trick here is that a Moore sentence is only true before the update, after the update it is false. Thus, an agent does not believe the statement anymore after an update (in our example, after learning p, it is the most plausible world, even if it wasn't before, which breaks the "NOT belief P" part), which causes the success axiom (you must come to believe what you learn) to fail. While a feature like this is not particularly relevant for an agent who is just concerned about reading sensor data, it is potentially useful for agents learning from each other. It can be argued that Moore sentences are somewhat "strange", in that you only start believing something if you didn't before. But it is certainly an interesting feature to have when agents are

communicating about higher order belief. For example "if agent 1 spoke to you, he lied about P" could yield the update "NOT P AND belief P", which is in turn a Moore sentence. So when talking about the believes of agents in this context, and agents can cheat or lie, a Moore sentence becomes quite useful.

So how does this technique hold up against the scenarios outlined in the research question? Public and truthful announcements are clearly covered. As we have seen, using relations between possible worlds to create plausibility models allows for ordering of worlds per agent, resulting in beliefs. For multi agent scenarios this can also be used to deal with private announcements. Finally, forms of false information can clearly be dealt with (as shown by the support for Moore sentences). It should be noted however that false updates will still lead to problems, only false upgrades can still yield proper results. In general, we can conclude that this technique is capable (in theory) of dealing with all three of our scenarios.

## *2.4 The final choice*

In the research question we posed three scenarios, summarizes as:

1. public truthful announcements
2. private truthful announcements
3. private false announcements

The first technique we looked at was Jeffrey Conditioning, which was found to be useful to model scenario 1, but had some issues with scenario 2 as there is no way to model private announcements, additionally, there is no obvious way to give less credibility to new information, as one can do with radical and conservative upgrades. Jeffrey conditioning by default only does radical upgrades. Finally, scenario 3 has a problem, as false information just cannot be modeled by this technique.

The second technique we looked at was Interpreted Systems, and a particular implementation of it. This technique was capable of modeling scenario 1. However, when using the implementation capable of dealing with false information (scenario 3), we lose the capability of dealing with scenario 1 properly. One of the problems we encounter is that this technique was originally made for measurement sequences, and determining how likely the next measurement is. A result of this is that the state tends not to change directly upon first encountering new information, while it is important that for example updates are carried out immediately. This makes this technique unusable for our intended scenarios.

The third and final technique we look at is Dynamic Epistemic Logic (DEL). This technique proves to be capable of dealing with all three of our scenarios using plausibility models containing belief relations on a per agent basis. Additionally, DEL also distinguishes between radical and conservative upgrades, which allows information sources to be trusted in lesser degree. While false announcements only require upgrades in general, having two levels of upgrades allows an agent to react differently to information from agents which are more trustable than others.

Concluding on this, only DEL was capable of dealing with our scenarios as outlined in the research question. The other techniques were capable of solving parts of the puzzle, but not everything. This is not to say the techniques are bad in general, they were made for different purposes, and do certainly have merit in their own way. This research does however show that they are not suitable for these specific kinds of scenarios.

# Chapter 3 - Practical work

In this chapter, we shall take a look at the practical part of our research questions. It is based on Dynamic Epistemic Logic (DEL) and this chapter expects the reader to have some knowledge of it. For more details on DEL we refer to the previous chapter, as well as (Baltag, Ditmarsch & Moss, 2008) and (Baltag & Smets, 2008). This chapter shows one possible solution to the problem, as is the way with writing software, there is more than one way to do it, and should provide pointers for anyone who wishes to retrace the steps taken to design the software that was made for this thesis. It does however not contain detailed instructions for each and every line of the code. For the full source code, see appendix B, for a manual containing instructions on how the software works, see appendix A. The code in appendix B is written entirely in Java and designed/tested for Java version 1.6 although it may also work on other versions. The reason this program was written in Java is that it is easily portable to both Linux, Unix and Windows systems, a capable programmer should however not have much problems implementing it in any other Object Oriented language. There are many reasons in favor or against using either Java or any other programming language, but all are outside the scope of this thesis. Finally, this chapter may refer to any classes contained in (Java SDK API, 2011), most notably *String*. These will be marked by the use of a capital letter, as well as an italic font. Any such classes can be either found in appendix B or in the Java SDK API.

Our program makes a few assumptions about how a formula should look like. These assumptions can be found in the manual in appendix B. Additionally, all parts of the syntax are encoded as reserved keywords and are user definable by use of a separate *config.ini* file which is read by the program at startup. The class *Util* is responsible for reading the config file and maintaining a list of all currently reserved keywords and their purpose. This way, all other parts of the program can make easy use of them, and it prevents us from requiring them being hardcoded. If we need to know if something matches for example a belief operator, we can just ask the *Util* class at runtime what a belief operator actually looks like.

One of the design choices we made was to completely split the user interface from the program itself. This way, if anyone wanted to just make another or more complex user interface, they would not need to know anything about how the rest of the program works, potentially saving a lot of time. Furthermore, it allows multiple user interfaces to be shipped with the program and used depending on the user's needs. Finally, it allows the user interface (or the program itself) to be changed without changing the program (or the user interface) at all, this is a limited form of the MVC pattern common in software engineering. A logical result from this is that all classes are made only accessible within the Java package they are defined in. The only class accessible from the outside, and providing all the functionality of this program to the user interface is Core, which is a typical case of both the Singleton and Facade patterns used in software engineering.

Additionally, we need a proper error reporting mechanism. In this particular case, we chose to use the *Exception* class, and in particular, instances of *RuntimeException* and custom classes inheriting from it. This is Java specific and eliminates the need to catch these exceptions at any level, instead, we can catch them if we want to add any information to the exception, and let it fly if we don't. At the top level, in this case the user interface, we can then catch all exceptions and hand the appropriate errors to the user.

The first step designing the actual model is "simple", we need a data structure that matches the theories about DEL. Since a plausibility model consists of worlds, agents and relations, this seems a good place to start. An agent is no more than a name label, which makes it easy to design, but raises the question if it should not be a regular *String*. The reason it was designed as a class is that we wanted to add some checks on what could be allowed as a valid agent name, in particular, we do not want the agent name to contain agent starting and ending tags. In addition, some comparison methods were added (based on those of a *String*) to allow quick comparison of multiple agents. The class *World* also relatively straightforward, only next to a name, a world also contains a list of propositions which are valid in this specific *World*. Checks must also be made to ensure a proposition name does not contain any reserved keyword used as part of the syntax of our formulas, if they did, they could create ambiguity in the formula, which is something we of course want to avoid. Lastly, we want to define the Relation class. A relation is nothing more and noting less than what its name suggests, and for our program it just need a reference to the *Agent* to which it belongs, as well as the *World* it came from, and the *World* it goes to.

To make matters a little bit easier to program, we define an *EquivalenceClass*. This class is based on its DEL counterpart, and contains a set of relations for one single agent. A set of relations is defined as all relations that can be accessed from a given world, and thus can be compared for belief. This also implies that the agent does not know the difference between all worlds thus related. It will have a preference for one or more worlds based on belief, but as far as knowledge is concerned, these worlds are indistinguishable for the agent. This will help us greatly resolve any belief or knowledge operators, since they always have a world (the real world at the start of any formula) from which they are posed. Using the *EquivalenceClass* we can quickly ascertain an agents beliefs or knowledge. When relations get added, they are also added to the corresponding equivalence class. If needed, the program will merge any such classes (when a relation is added that forms a link between two equivalence classes). Equivalence classes can only contain unique relations, and use these relations to (to put it plainly) count arrows going to each world to determine their belief order.

These four classes in turn are part of larger entity, namely a state. The *State* is responsible for collecting all worlds, agents, relations and equivalence classes, so it can add them, remove them and list them. In addition, the *State* is responsible for performing public and private updates and upgrades (radical and

conservative). To do this, the *State* will be given a list of worlds in which the update or upgrade is true, and for which agents. The actual evaluation is done in another class, which we shall discuss in the next paragraph.
Public updates are relatively easy, the State just needs to drop all relations to and from the world(s) in which the update was true and those in which it was false, while preserving the relations between all true world, and all false worlds.
Private updates are different, as some agents will be unaware it has been made, so we need to make a duplicate of all relations to ensure for them everything stays the same. Then, from the duplicate set of relations we remove the relations going to or from false worlds. After that, we make a copy of all true worlds (prefixing the names with unique numbers, just as one would add an accent character in any of the papers dealing in event models) and we modify the duplicate relations to use the duplicate worlds instead. We also add a relation from each duplicate world going back to their corresponding originals for agents who were unaware of the update. We also need all relations from the true worlds to the false worlds to exist between the duplicate worlds and the original worlds for all agents that were not aware of this update. Finally, to keep the connected property, if there was a relation from a false world to a true world for an agent who did not know about the update, we also need to add a relation from the duplicate of the world it was going to, to the original the relation was coming from.
Public radical upgrades are easy too, we need to promote all worlds in which the statement is true to become the most plausible worlds, which means we need to reverse all relations going away from those worlds (without creating duplicate of course).
Conservative upgrades can be dealt with in identical fashion.
Private radical upgrades are a bit more complex. Just as with the private updates, duplicates are required, only this time of all worlds and relations. The changes like for the public radical upgrades are then made on the duplicates for all agents aware of the upgrade. Additionally, for each agents not aware of the upgrade, a relation should be added from each duplicate world, to its corresponding original world. Finally, all relations for all agents not aware of the upgrade should be copied and made from each duplicate world corresponding to the from world of the relation to the original "to" world of the relation.
Private conservative upgrades are not present at all, due to them not existing yet in the literature. This is because in the current setting, with a given restricted syntax, there is no matching event model for them. The attentive reader would point out that the above do also not use an event model. This is because this program performs updates and upgrades based on the behavior of one specific event model, namely the one which contains only one event known to one group of agents. This is because the author believes this one event model can be used to model (almost) any situation except perhaps simultaneous private upgrades or updates. In all situations one may wish to model, they do not generally need to be simultaneous. For more information about this issue, see the Discussion section of this paper. It does however imply that it should be possible to make a private conservative upgrade this way, since this program copies the behavior of one specific event model rather than using the actual event models. However, since there is no proper research done yet on this issue, and it is considered outside the

scope of this paper, no attempts have been made to add this feature to the program.

The *Evaluator* is the class responsible for parsing any formulas a user may wish to evaluate. To do so, it contains a method which recursively calls itself, each time making the formula smaller by taking out one operator until in the end it is faced with one single proposition, to which it can assign a truth valuation, which is then returned to help evaluate the previous operator. Taking such an approach leads to a dealing with any formula as a tree, which is then searched depth first. Because of this, memory use scales linear to the amount of operators rather than quadratic which would happen when one evaluates it width first. The actual evaluation time is still quadratic to the amount of operators, but the memory usage makes it manageable for modern day computers, which typically have limited memory, and unlimited time (well, years actually, not truly unlimited time).
Before the recursive method is called, first the program checks for a turn-style to determine on which world the formula is to be evaluated. When there is no world before the turn-style, the formula is evaluated on all worlds. The recursive method then checks for parenthesis. If no operators are outside them, the program will try to remove the outermost parenthesis. Next it will search for binary operators, in order from the lowest priority to the highest according to regular propositional logic. The exact order (from low to high) is: EQUALS, IMPLIES, OR and AND. Finally, it will seek out any unary operators (NOT, belief, conditional belief, knowledge, update, upgrade). Dealing with the operators from propositional logic is trivial. Knowledge will be determined as "must be true on all worlds in the same equivalence class of this agent at the world we currently evaluate on". Belief is evaluated as "must be true on the most believed world in the equivalence class for this agent belonging to the world we evaluate on". In case of conditional belief, it is evaluated on the most believed world for an agent on which the condition still holds. The *EquivalenceClass* is used here to determine the actual equivalence classes, as well as provide the belief order.
For any of the update or upgrade operators, the *Evaluator* first makes a copy of the current *State*. This to be able to revert it later for different parts of the evaluation tree, or later evaluations. Then the worlds on which the formula holds are determined by again recursively calling the same method, but this time to evaluate the update or upgrade condition on each world to see on which it is true. The actual upgrades and updates are then performed by the *State* as mentioned earlier. After which the rest of the formula is evaluated on this changed *State*.

With the whole framework taken care of, there is one last finishing touch, a data file to read our initial plausibility model from. We could let the user manually input all relations, agents and worlds each time, but it is a lot easier to just put them in a data file. So, we also added a data file reader. It works fairly straightforward, and most complexity goes into checking for reflexive relations and bad input (invalid names, duplicate names, etc). The class is called *BelieverFileReader*, since it is quite separate from the rest of the program, it can

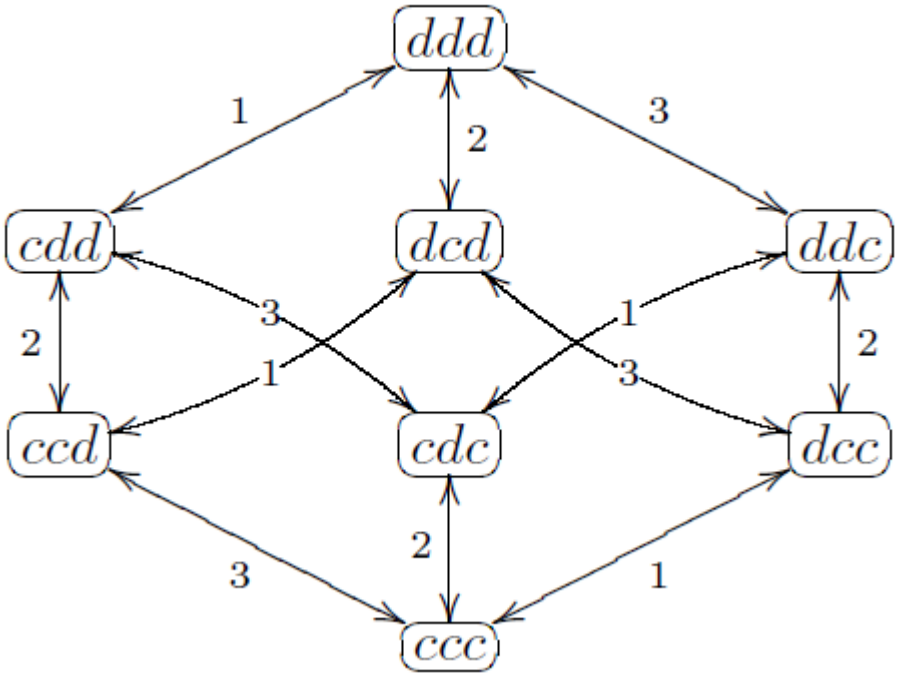
easily be changed or replaced to support another format. For details on our data format, we refer to the manual in appendix A, and to appendix B for our actual implementation of this file.
Another thing not mentioned explicitly in the description of any classes is error checking, it is there, but it is not relevant for being able to recreate this program (if it is done right). Additionally, a so called JUnit test was added for automated testing, this class is *TestCore*, which uses the *Core* class to test correct evaluation of various logical formulas on a predefined test *State* which it also creates for this purpose. Again, we won't go into details of this class due to it not being required for this program to work, but we do list it here for completeness, as it gives an indication of what this program was actually checked to be capable of.

This concludes the overview of how all classes are constructed and how and why they were designed. For discussion on some points of the design we refer to the Discussion section, and finally, for future recommendations we refer to the respective chapter.

# Chapter 4 - Results

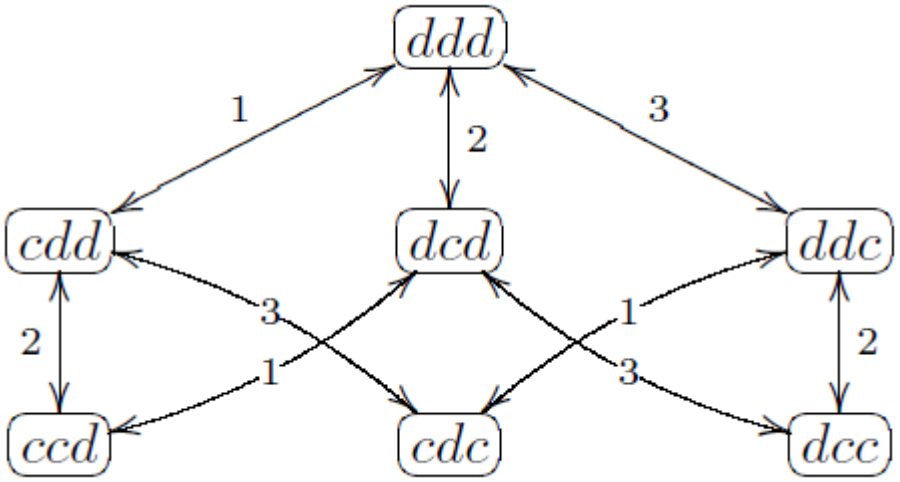
In this chapter, we shall use the program (see appendix B) we developed, to solve the muddy children puzzle to show it can indeed properly handle DEL plausibility models as well as to prove its potential use in similar applications. The images in this chapter were taken from a course called Multi-Agent Belief Dynamics by A. Baltag and S. Smets. Making a program like this shows that DEL plausibility models in combination with event or action models in some form can be automated. The potential uses for this are many, among others: robotics, computer games, analyzing politics and AI in general.

The muddy children puzzle is one of the problems commonly used in logic to show the importance of belief revision, and to prove a certain model can indeed properly revise its beliefs. Alternate versions of this puzzle are Cheating Husbands Problem, the Unfaithful Wives Problem or Josephine's Problem (and probably a few more). In this particular case, there are *n* children who have been playing outside, when they get back home, their father sees *m* of them have mud on their face. These children however are special children, in that they are perfect logicians and perfectly honest. In addition, they can see each other so they know who of the others is muddy, but they cannot see their own face, so they do not know if they are muddy themselves. So the DEL plausibility model for this situation with three agents will look like this (notice that the reflexive relations have been omitted from this image, they do however exist):

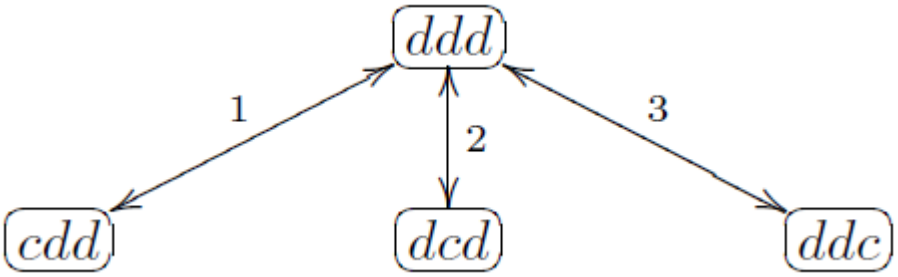



Where "c" means a child is clean, and "d" means a child is dirty, and the arrows show a certain agents belief.

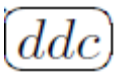
To start this puzzle, the father announces to the children "at least one of you is muddy". This sentence has special importance, since it not only tells the children that the world in which they are all clean is not the case, it also tells them that all the other children will now know at least one is dirty. Thus, it becomes common knowledge. While it cannot be explicitly modeled in our language, it is implied, and essential in solving this puzzle. All other updates in this puzzle are likewise common knowledge. The fathers announcement resulted in the following model:




The father then asks "if you know you are muddy or clean, please say so now". After this question, if there was only one muddy child, he will know (after all, he cannot see anyone who is muddy, so he knows it must be him). When there are more children muddy, none will act. This actually changes the model, as now all agents know there is at least one more muddy child, thus invalidating all worlds in which only one child was dirty. This changes the model again:




After the children have answered, the father will repeat his question over and over again, until all the children have answered. If "ddc" is our true world, agent 1 and 2 will know they are clean (they have no relations going away from the "ddc" world, and thus they only believe that one). They thus answer their fathers question by "yes, we are dirty" while the third child still does not know. This changes the model again to just:



Now the third child knows too, so when the father asks his question for the third time, he shall answer "I am clean", thus ending the puzzle.
It can be noted that for this puzzle, the amount of worlds equals $2^n$ and the children who are muddy will always know that they are after the father asks his question exactly *m* times.
So now we have seen what the puzzle is, it is time to see if our program can solve it. For this we have to encode the original puzzle to an input file for our program (see appendix C for this file). Once we have this file, we shall illustrate first how we can make this work with true, infallible announcements, as they are used commonly with this puzzle. Afterwards, we shall show this model can also deal with lower degrees of trust, and even cheating children. The full commands as given to the program are listed here as well, so this example can be easily reproduced and checked. For details on the full syntax used, or how to operate the program, see appendix A. For this example we shall use the following operators: K{a1}C1 meaning agent 1 (a1) knows that agents 1 is clean (C1), similarly B{a1}C1 means agent 1 believes agent 1 is clean. The standard logical operators ^ (AND) and v (OR) and ~ (NOT) are straight forward too. UPDATE(C1) is a public update with C1, UPDATE{a1}(C1) is a private update for agent 1 that C1 is true. Similarly, RADICAL_UPGRADE{a1}(C1) is a private radical upgrade for agent 1 saying C1 is true. Radical upgrades promote all worlds in which something was true to be the most believed, in this example, conservative upgrades (only promoting the most believed world matching the upgrade) would have yielded identical results.

Once we have started our program, we need to load the file by typing:

load file path_to_file/filename

In this example, we shall assume world 6 (w6) is the real world, thus all evaluations shall be done there. We start by checking that none of the agents knows if he is clean or not:

w6 |= K{a1}C1 v K{a1}~C1 v K{a2}C2 v K{a2}~C2 v K{a3}C3 v K{a3}~C3

This indeed yields false. The agents can however see each other, so if we ask one agent if the other is clean, we will get a result:

w6 |= K{a1}C2

Showing that agent 1 indeed knows that agent 2 is clean (C2), this yields true. Any agent can determine the cleanliness of any other agent this way.
Next, we want to encode the fathers statement "at least one of you is dirty". This is the equivalent of saying "the world in which you believe all are clean is not true" or UPDATE(~(C1^C2^C3)). After this statement, since there are 2 agents dirty, agent 1 still does not know if he is clean or not:

w6 |= UPDATE(~(C1^C2^C3)) (K{a1}C1 v K{a1}~C1)

Which indeed results in false again. This also results in false for all the other agents. So none of the children steps up when the father asks " if you know you are muddy or clean, please say so now". This is the same as saying that all children do not know if they are dirty or clean ((~K{a1}C1 ^ ~K{a1}~C1) ^ (~K{a2}C2 ^ ~K{a2}~C2) ^ (~K{a3}C3 ^ ~K{a3}~C3) ). Notice that while the first update was made up of non epistemic facts, this second update states just what the children know. After this statement however, agent 1 and 3 do know the answer:

w6 |= UPDATE(~(C1^C2^C3)) UPDATE((~K{a1}C1 ^ ~K{a1}~C1) ^ (~K{a2}C2 ^ ~K{a2}~C2) ^ (~K{a3}C3 ^ ~K{a3}~C3)) K{a1}~C1

This results indeed in true. Agent two however still does not know, since clean children are always last to learn that fact as we've seen earlier in the puzzles explanation. This is show by:

w6 |= UPDATE(~(C1^C2^C3)) UPDATE((~K{a1}C1 ^ ~K{a1}~C1) ^ (~K{a2}C2 ^ ~K{a2}~C2) ^ (~K{a3}C3 ^ ~K{a3}~C3)) (K{a2}C2 v K{a2}~C2)

Once the father asks his question for the second time, the children 1 and 3 now know the answer and say so. From this, agent 3 learns that that the others know, and after an update using that information, he knows too:

w6 |= UPDATE(~(C1^C2^C3)) UPDATE((~K{a1}C1 ^ ~K{a1}~C1) ^ (~K{a2}C2 ^ ~K{a2}~C2) ^ (~K{a3}C3 ^ ~K{a3}~C3)) UPDATE(K{a1}~C1 ^ K{a3}~C3) K{a2}C2

And so agent 2 can answer the fathers question as well the third time he asks it.

This shows that the program can indeed deal with the original muddy children puzzle. But what if the children know that they are not infallible? Now they will have to resort to belief instead of knowledge. As it turns out, any of the above statements will still hold if K is changed for B and UPDATE is changed for RADICAL_UPGRADE. We shall not repeat all statements, but let us take a look at the situation when the children have just passed the fathers question for the first time. Indeed, agent 1 again believes he knows the answer:

w6 |= RADICAL_UPGRADE(~(C1^C2^C3)) RADICAL_UPGRADE((~B{a1}C1 ^ ~B{a1}~C1) ^ (~B{a2}C2 ^ ~B{a2}~C2) ^ (~B{a3}C3 ^ ~B{a3}~C3)) B{a1}~C1

And agent 2 indeed  does neither believe he is clean, nor that he is dirty:

w6 |= RADICAL_UPGRADE(~(C1^C2^C3))
RADICAL_UPGRADE((~B{a1}C1 ^ ~B{a1}~C1) ^ (~B{a2}C2 ^
~B{a2}~C2) ^ (~B{a3}C3 ^ ~B{a3}~C3)) (B{a2}C2 v B{a2}~C2)

Once he hears the first two children say they are clean after the fathers second question, agent 2 knows that can only happen when they see he is clean. But since he also knows they are not infallible, he cannot make an update and he cannot be completely sure of their observations or their logic either, so all he knows is that they believe they are dirty, and from updating the model with that information he will now believe he is clean:

w6 |= RADICAL_UPGRADE(~(C1^C2^C3))
RADICAL_UPGRADE((~B{a1}C1 ^ ~B{a1}~C1) ^ (~B{a2}C2 ^
~B{a2}~C2) ^ (~B{a3}C3 ^ ~B{a3}~C3)) RADICAL_UPGRADE(B{a1}~C1
^ B{a3}~C3) B{a2}C2

This proves that the program can also solve the puzzle when the agents are not infallible. But what would happen if one of the children cheats? Let us say agent 1 takes a look in the mirror after the father announces at least one child is dirty (RADICAL_UPGRADE{a1}(~C1)). He believes he is dirty (he does not know since he's not infallible himself either):

w6 |= RADICAL_UPGRADE(~(C1^C2^C3))
RADICAL_UPGRADE{a1}(~C1) (B{a1}~C1)

Agent 2 and 3 do however still not know if they are dirty or not:

w6 |= RADICAL_UPGRADE(~(C1^C2^C3))
RADICAL_UPGRADE{a1}(~C1) (B{a2}C2 v B{a2}~C2 v B{a3}C3 v
B{a3}~C3)

What is more, they are also unaware agent 1 has cheated (since they did not see him look in the mirror), we only show this for agent 2, but the same is true for agent3:

w6 |= RADICAL_UPGRADE(~(C1^C2^C3))
RADICAL_UPGRADE{a1}(~C1) (B{a2}B{a1}~C1 v B{a2}B{a1}C1)

So when he announces he is dirty when the father first asks his question, agent 2 will know something is wrong, after all, he can see two muddy children in front of him. He just does not know if he himself is dirty or clean. Agent three however will be deceived and tricked into believing he is clean. Since he is a perfect (but not infallible) logician, he believes agent 1 can only reach that conclusion if he sees two clean agents or if agent 1 cheats, but he prefers to believe agent 1. Still,

because it is possible and the agents are wrong, or lying, beliefs and radical upgrades are used rather than knowledge and updates:

w6 |= RADICAL_UPGRADE(~(C1^C2^C3))
RADICAL_UPGRADE{a1}(~C1) RADICAL_UPGRADE(B{a1}C3 ^
B{a1}C2) B{a3}C3

Agent 2, who already knew that at least two agents are muddy, is not affected by this upgrade, he still doesn't know:

w6 |= RADICAL_UPGRADE(~(C1^C2^C3))
RADICAL_UPGRADE{a1}(~C1) RADICAL_UPGRADE(B{a1}C3 ^
B{a1}C2)(B{a2}C2 v B{a2}~C2)

However, when the father asks his question for the second time, agent 3 will say he is clean (falsely so). Agent 2 will now be able to conclude that the statement by agent 1 was achieved by cheating, and that agent 3 was mislead by this. He now also realizes that this could only have happened if agent 2 is clean, and so agent 3 only saw one other dirty agent. Because if they had both been dirty, agent 3 would have seen this, and would not have said he was clean. They would both have believed they were dirty after the father asked the question for the third time
So agent 2 now believes he is clean as show by:

w6 |= RADICAL_UPGRADE(~(C1^C2^C3))
RADICAL_UPGRADE{a1}(~C1) RADICAL_UPGRADE(B{a1}C3 ^
B{a1}C2) RADICAL_UPGRADE(B{a3}C2) B{a2}C2

Which proves that our program can also deal with cheating children, and that some agents can even find out about this cheating and resume normal operation afterwards.

If agent 1 had instead of cheating resorted to lying by claiming he was clean, it would in this case have been obvious to agent 2 he lied. Agent 3 however would not know if he was lying or cheating. In both cases, agent 3 would see that agent 2 did come forward when the father asked his question the second time. This would tell agent 3 that he must be dirty too (or agent 2 is cheating or lying as much as agent 1). Once he uses this information to claim he is dirty, agent 3 will claim he is clean too after the fourth question. So while this deliberate lying delays the outcome by requiring the father to ask his question once more, the other two agents are eventually unaffected by it and still solve the puzzle (if they both are honest).

We shall leave the actual statements to the curious reader, but do point out that in our cheating example, the upgrade RADICAL_UPGRADE(B{a1}C3 ^ B{a1}C2) is actually a lie too (in disguise). The agents (agent 3 in particular) assume this

statement is true, but it isn't. In the real world, agent 1 believes agent 3 is actually dirty, or more formally the following is true:

w6 |= RADICAL_UPGRADE(~(C1^C2^C3))
RADICAL_UPGRADE{a1}(~C1) B{a1}~C3

The agents are thus revising with false information. To be technically correct, agent 2 knows this, and one could decide to instead model this upgrade as a private upgrade just for agent 3. The outcome would be the same though. This shows that our program can handle both public announcements, private announcements and cheating and lying.

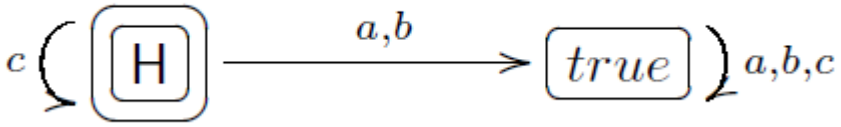
# Chapter 5 - Discussion

As with any research, there are some points that may seem to be strange choices, or are otherwise arguable. In this section, we will list any such points we could find, as well as the reasoning for doing it this way.

Research questions: The practical part of the research question seems to be prejudiced in terms of which of the three techniques discussed in the theoretical part is the most suitable for the practical part. While it is true that at the start of this research DEL based plausibility models were chosen as the main technique to do the practical part with, it did not influence the outcome of our comparison between the three techniques. Arguably, the question should have been posed differently, but since parts of the theoretical and practical work were done at the same time, it was chosen to pose the question as it is, with all presumptions.

Another thing that could be called into question is the choice of these 3 techniques. Why not use AGM (Alchourrón, Gärdenfors, and Makinson, 1985)? Or other techniques? There are simply too much different techniques out there to consider them all in this thesis, so instead these three were chosen based on what the author believes is their being representative for a whole group of different techniques. It may be noted that Interpreted systems are largely compatible with AGM, and the Jeffrey Conditioning is based on Bayes theory, and could be considered similar in technique and applicability to a techniques based on Markov Chains (Markov, 1971). There are of course some differences, and this article does not claim them to be identical, just similar enough for the purposes of this article to not consider them separately.

The use of event models (Baltag & Smets, 2008) by the program which was designed as part of this research is, as mentioned in chapter 3, not literally implemented. In particular, the program mimics the behavior of one specific event model:




In this model, we see on the left a certain action (in this case H, but it could have been any formula), and on the right the "true" action, to indicate everything stayed the same. In this specific event model, agent C knows action H happened, while agent A and B think nothing has happened. The actual agents in this example are not what the program mimics, instead it mimics the fact that one event happens for some agents, and nothing happens for some others, and combines this with update and upgrade statements. If all know the event happened, the results are the same as for a public announcement.

The reason we let the program mimic the behavior for this event model, rather than letting the user specify a precise model is twofold. On one hand, it would require the user of the program to specify all desired event models in advance, which is much work and not very dynamic, on the other hand, this specific event model can be used in combination with updates to model anything an event model could. Proof on event models being capable of dealing with any update or vice versa can be found in (Baltag, Moss & Solecki, 1998).
At the moment, not much research has yet been done (to the author's knowledge) on the differences between event models and update statements. So our current implementation seems to be the most user friendly solution, even though it might take a user who normally uses event models a few moments to adapt these to update statements. The only alternative would be to implement both event models and update and upgrade statements, and using them simultaneously, which was deemed not very user friendly.

# Chapter 6 - Conclusion & future recommendations

In this chapter, we shall sum up the results of all previous sections. Additionally, we shall make some recommendations for possible future work.

## *6.1 Conclusion*

The second chapter showed that there are various differences between Jeffrey Conditioning, Interpreted Systems and Dynamic Epistemic Logic (DEL). Some of which would interfere with the requirements set in our research question in chapter 1.1. It turned out that only could deal with both public truthful updates and upgrades, public updates and upgrades of less trusted sources, private updates and upgrades (cheating) and outright lying agents all at the same time. Some of the others could do a few of these depending on the exact implementation, but not in one version.
Chapter 3 then discussed how a program could be made to deal with DEL and a form of event models to provide automated handling iterated updates and upgrades, while of course still supporting regular propositional logic. In chapter 4 we showed that this program was indeed, capable of dealing with public and private truthful and untruthful updates and announcements. This proved that our program had indeed the same capabilities as one would expect a DEL model to have when combined with event models.

## *6.2 Future Recommendations:*

While both the research and the program are sound, there are always more things to do, and different ways to do it. In this section we shall provide a short list of recommendations for future work.

1) Currently, the program is working in a single threaded fashion. Needless to say, with the advance of dual-, quad- and even octa-core processors in regular desktop computers, and the expectation that this trend will continue for a while in the near future, it would be beneficial for more complex models to be able to make use of multiple processors. The most straight forward way would be to make new threads each time the program encounters a binary operator, and have one thread work out each side of it. However, to make that work properly with optimizations such as "short circuiting" the operators would require more work, and likely more research as to how this can be done efficiently. Especially as one will most likely not want to make new threads on every binary operator due to the exponential amount of threads that would create. Additionally, the program in its current form could likely be optimized a little bit more.
2) The command line interface currently provided is of course sufficient to operate the program, and execute any formula imaginable. It may however be desirable to be able to use a Graphical User Interface (GUI) to perform some cosmetic upgrades on the formulas, like highlighting matching

brackets, or hiding and un-hiding the content between them, to make complex formulas easier to read. A GUI display for the current model, and an option to edit them graphically would also greatly improve this programs usability with more complex models.

3) Formula checking is something that is currently done largely on the fly. This means that an incorrect formula may not result in any error for many hours, or even days in case of a really complex formula. So formula checking could be added as an option to ensure these situations do not arise.
4) The models currently rely in part on valid user input. Some more complex checks on transitivity, reflexivity and connectivity of the relations could make it impossible to create invalid models. This option should preferably be combined with the addition of a GUI, so that errors can be clearly highlighted, and recommendations can be offered in an interactive way.
5) Private conservative upgrades are not implemented yet, however it is believed that they should be possible. This is due to the fact that this program does not use Event Models, but allows for formulas instead. A private conservative upgrade is (even for multiple agents) nothing more than a simple formula if not Event models are used. Of course it does still need implementation.
6) A point of theoretical interest that is closely related to the previous recommendation is proving that Event Models can be used to yield identical results as formulas, and similarly, that formulas can be used to express all possible Event Models. Or, depending on the outcome of such research, proving that they can actually not be used to represent the same thing under some circumstances.
7) While DEL based plausibility models do offer a lot of options in terms of belief change, it currently lacks support for factual change. DEL with factual change has been studied in detail before, the setting of plausibility models in DEL however has not yet been explored in connection to this. Technically the same techniques for factual change could also be used in the plausibility model context (such as "substitution of sentences"). This would allow us to explore situations such as the following: if an agent were to learn that for example there is no more ice on the south pole? It could be dealt with as a belief revision, but only if the model already contained a proposition for this condition. But an agent can also encounter things not imagined when the agent was originally programmed. So ideally, one would want a possibility to make a statement like "LEARN(phi)". This way, an agent would be using belief revision in a changing world, which is very usable in our real world.

# Appendix A: Manual

**getting started**
To start the program, run the believer.jar file. How this can be done depends on the operating system. On a command shell (DOS shell in windows), it can be done by using the command "java -jar believer.jar". Additionally, windows users can just double click the believer.bat file, which does it automatically for them.

**reading models**
Once the program is running, a DEL model needs to be loaded by typing "load file path_to_file/file_name". The file consists of a list of agents, a list of worlds and the propositions which are true on those world, and a list of relations. In the file ExampleInputFile more detailed instructions are given on the format, as well as an example. Note that all relations must be reflexive and transitive. Furthermore, all relations which are connected must have an order. That is to say, they must be comparable in terms of which is the most believed. Making a model like:
A <- B -> C
Is not allowed, as in this model, both C and A are more believed than B, but they are not equivalent, and are not comparable either. (Neither A nor B is most believed. Adding a relation from A to C, or vice versa will solve this, and make the model valid again.

**list**
After a file has been read, the commands "list agents", "list relations", "list worlds" and "list equivalence classes" can be used to show that all the agents, relations, worlds and equivalence classes as the program has read them. This is mostly useful if a formula yields unexpected results, or to ensure a file was read properly.

**evaluate**
The most used command will be the "evaluate" command, which is followed by a formula. Which syntax a formula normally has is determined by the contents of the config.ini file. Most commands there will look familiar to any logician, but a few to note are "agent_start_tag (and agent_end_tag), which are basically the enclosing tags for an agent name. By default they are { and } respectively, so a belief operator (default B) would look like B{some_agent}. The agent_seperator (,)is literally that. If one wants to use multiple agents, one would get B{agent1, agent2}. The "conditional" is used for conditional beliefs, so it would be used as B{agent1}CONDITIONAL(Q) P, where it means that an agent1 believes P conditional on Q. Finally, the various update and upgrade statements all work the same, and have the form of "UPDATE(P) Q" update with P, then evaluate Q, or in private form "UPDATE{agent1}(P) Q, where it will only update P for agent 1 as a private update and then evaluates Q again.
All formulas start with the turn-style (default |= ), which is used to determine the real world, which should be in front of it. If no world is provided in front of it, the program will evaluate the formula on all worlds.

The default values for AND, OR, NOT, IMPLIES, EQUALS and knowledge are (in order) ^,  v , ~, ->, <=> and K. The OR is a little bit special in that it is a lower case letter v, with a space in front and one behind it. This is to ensure the letter v itself does not become a reserved keyword. It can however be changed to anything (including only the letter v) in config.ini, if a user considers this more convenient. The config.ini file is read automatically when the program starts.

**verbose**

The command "verbose on" (and to turn it off again, "verbose off"), tells the program to display additional information about which choices it has made during evaluation of a formula, and what the result of those choices is. It is debug information, which should probably be left alone until a formula starts to show unexpected results and the model is certain to be correct.

**the others**

The last commands are "help", "quit" and "exit". Quit and exit do exactly the same, and exactly what one would expect. The help command displays a list of all available commands and a quick one liner on what they do. This is mainly helpful if you can't remember how to type a command, more detailed information is in this manual.

# Appendix B: Source code

A compiled version of this program, as well as an SVN version of this source code can be found on http://code.google.com/p/believer/ for as long as the host (Google) decides to host the page.

This appendix contains the following files in order:
nl/rug/ai/believer/model/Agent.java
nl/rug/ai/believer/model/BelieverFileReader.java
nl/rug/ai/believer/model/Core.java
nl/rug/ai/believer/model/EquivalenceClass.java
nl/rug/ai/believer/model/Evaluator.java
nl/rug/ai/believer/model/Relation.java
nl/rug/ai/believer/model/State.java
nl/rug/ai/believer/model/Util.java
nl/rug/ai/believer/model/World.java
nl/rug/ai/believer/model/exception/DuplicateException.java
nl/rug/ai/believer/model/exception/InvalidAgentException.java
nl/rug/ai/believer/model/exception/InvalidWorldException.java
nl/rug/ai/believer/ui/CommandLineInterface.java

Paths are due to Java packaging conventions. The main point of entry for the program is in "nl/rug/ai/believer/ui/CommandLineInterface.java"

Files start at the beginning of a page. No two files are listed on the same page.

## nl/rug/ai/believer/model/Agent.java

```
package nl.rug.ai.believer.model;

/**
 * This class represents an agent. Agents only have a name, which is just a
label for human users to
 * easily identify a certain agent.
 * @author S1849700
 *
 */
public class Agent {

	private String name;

	/**
	 * Standard constructor, initializes the agent with a name.
	 * @param name the name of the agent
	 */
	public Agent(String name) {
		if (name == null) {
			throw new NullPointerException("Constructor name of Agent
was null");
		}
		if (name.indexOf(Util.agentSeparator) != -1 ||
name.indexOf(Util.agentEndTag) != -1 ||
				name.indexOf(Util.agentStartTag) != -1) {
			throw new RuntimeException("Agent: \"" + name +
"\"contains a reserved keyword.");
		}
		this.name = name;
	}

	/**
	 * accesor method for the name
	 * @return the name of the agent.
	 */
	public String getName() {
		return name;
	}


	/**
	 * We consider an Agent equal if their name is equal.
	 * @see Object#equals(Object)
	 */
	@Override
	public boolean equals(Object other) {
	    //check for self-comparison
	    if ( this == other ) {
	      return true;
	    }
	    // this renders an explicit check for "other == null" redundant,
since it does the check for
	    // null already - "null instanceof [type]" always returns false.
```

```
            if (!(other instanceof Agent)) {
               return false;
            }
            Agent otherAgent = (Agent)other;
            return otherAgent.name.equals(name);
        }

        /**
         * @see Object#hashCode(Object)
         */
        // must be overwritten if .equals is overwritten. (And should be based
on the same key fields.)
        @Override
        public int hashCode() {
               return name.hashCode();
        }
}
```

## nl/rug/ai/believer/model/BelieverFileReader.java

```
package nl.rug.ai.believer.model;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

/**
 * This class is responsible for reading a file, and extracting all
information from it (to be used
 * by the Believer program).
 * @author S1849700
 *
 */
public class BelieverFileReader {

	private State state;
	private State realState;

	/**
	 * Main method for this class, used to read a certain file, and put its
contents into a state.
	 * When the reading was successful, the state in the parameter will be
overwritten with the
	 * new content. If unsuccessful, the state in the parameter will remain
the same.
	 * True or false is returned to indicate if the file was read
successfully. Note that this class
	 * will also do some sanity checks on the input.
	 * @param file - the file to read
	 * @param realState  - the state into which to write the file if it was
read succesfully.
	 * @return -- true or false, depending on the success of reading the
file.
	 */
	boolean read(String file, State realState) {
		state = new State();
		this.realState = realState;
		String input = getStringFromFile(file);
		String temp[] = input.split("AGENTS"); // ignore all before this
marker, so we only use [1]
		if (temp.length != 2) {
			Util.debug("Found a bad number of AGENTS tags");
			return false;
		}
		temp = temp[1].split("WORLDS");
		if (temp.length != 2) {
			Util.debug("Found a bad number of WORLDS tags");
			return false;
		}
		String agents = temp[0];
		temp = temp[1].split("RELATIONS");
```

```
            if (temp.length != 2) {
                  Util.debug("Found a bad number of RELATIONS tags");
                  return false;
            }
            String worlds = temp[0];
            String relations = temp[1];
            try {
                  if (extractAgents(agents) && extractWorlds(worlds) &&
extractRelations(relations)) {
                        this.realState.copyFromOther(state);
                        return true;
                  }
            }
            catch (RuntimeException rte) {
                  throw new RuntimeException("While trying to read input
file: " + file + "\n" +
                              rte.getMessage());
            }
            return false;
      }

      /**
       * This method extracts the contents of a file and puts it into a single
string.
       * @param file The file to be read
       * @return The resulting string
       */
      private String getStringFromFile(String file) {
            StringBuffer buffer = new StringBuffer();
            try {
                  Util.debug("path is: " +((new
File(".")).getCanonicalPath()));
                  FileReader fr = new FileReader(file);
                  BufferedReader br = new BufferedReader(fr);
                  int character;
                  while ((character = br.read()) != -1) {
                        // if we find a comment line, skip it.
                        if (buffer.length() != 0 &&
buffer.charAt(buffer.length() - 1) == '\n' && (char)character == '#') {
                              br.readLine();
                              continue;
                        }
                        buffer.append((char)character);
                  }
                  Util.debug("I now read: " +buffer.toString());
            }
            catch (IOException ioe) {
                  Util.debug("error reading file: " + ioe.getMessage());
            }
            return buffer.toString();
      }

      /**
       * Extracts all agents from a string and puts them into agentList
       * @param agents the string
```

```
	 * @return the success of the action
	 */
	private boolean extractAgents(String agents) {
		Util.debug("Received agents: " + agents);
		String temp = agents.replace('\n', ' ');
		String myAgents[] = temp.split(",");
		for (String agent : myAgents) {
			if (agent.trim() == "") {
				Util.debug("Encountered an empty agent string");
				return false;
			}
			try {
				state.addAgent(new Agent(agent.trim()));
			}
			catch (RuntimeException rte) {
				throw new RuntimeException("while reading agents:
\n" + rte.getMessage());
			}
		}
		return true;
	}

	/**
	 * Extracts all worlds and their values from a String, and puts them
into worldList
	 * @param worlds The String to be searched
	 * @return The success of the extraction
	 */
	private boolean extractWorlds(String worlds) {
		Util.debug("Received worlds: " + worlds);
		String temp = worlds.replace('\n', ' ');
		String myWorlds[] = temp.split("[()]");
		for (int  i = 0; i + 1 < myWorlds.length; i += 2) {
			if (myWorlds[i].trim() == "") {
				Util.debug("Encountered an empty world name");
				return false;
			}
			World world = new World(myWorlds[i].trim());
			if (myWorlds[i+1].trim() == "") {
				continue; //no values, don't try to add them.
			}
			String values[] = myWorlds[i+1].split(",");
			for (String value : values) {
				if (value.trim() == "") {
					Util.debug("Encountered an empty value for
world: " + world.getName());
					return false;
				}
				try {
					world.addProposition(value.trim());
				}
				catch (RuntimeException rte) {
					throw new RuntimeException("While trying to
add \"" + value.trim() +
```

```
                                                "\" to Wold: \"" +
world.getName() + "\"\n" + rte.getMessage());
                                }
                        }
                        state.addWorld(world);
                }
                return true;
        }

        /**
         * Extract all relations from a String, relations are grouped per agent.
Then add them to relationList.
         * @param relations The String to be read
         * @return the success of this operation
         */
        private boolean extractRelations(String relations) {
                Util.debug("Received relations: " + relations);
                String lines[] = relations.split("\n");
                Agent agent = null;
                World toWorld[] = null;
                for (String line : lines) {
                        // empty lines exist between entries only.
                        if (line.trim().equals("")) {
                                agent = null;
                                toWorld = null;
                                continue;
                        }
                        // only if there is no agent are we expecting the next
(non empty) line to be an agent name
                        // an implication of this is that the entire block will be
skipped if there is no such agent.
                        if (agent == null) {
                                agent = state.getAgent(line.trim());
                                continue;
                        }
                        // fill "toWorld" array if it is null (logically the next
line after the agent)
                        if (toWorld == null) {
                                String worlds[] = line.split(",");
                                toWorld = new World[worlds.length];
                                for (int i = 0; i < worlds.length; i++) {
                                        toWorld[i] =
state.getWorld(worlds[i].trim());
                                        if (toWorld[i] == null) {
                                                Util.debug("Can not find a to world
named: " + worlds[i] + "for agent: " + agent.getName());
                                                return false;
                                        }
                                }
                                // Add a reflexive relation for every world not
included for this agent in the input file.
                                ArrayList<World> allWorlds = state.getWorlds();
                                for (World myWorld : toWorld) {
                                        allWorlds.remove(myWorld);
                                }
```

```
                        for (World allWorld : allWorlds) {
                              state.addRelation(new Relation(agent,
allWorld, allWorld));
                        }
                        continue;
                  }

                  // each (non empty) next line is of the form "fromWorld,
relation, relation.... "
                  String words[] = line.split(",");
                  World fromWorld = state.getWorld(words[0].trim());
                  if (fromWorld == null) {
                        Util.debug("Can not find a from world named: " +
words[0] + "for agent: " + agent.getName());
                        return false;
                  }
                  if (words.length - 1 != toWorld.length) {
                        Util.debug("mismatch on entries on for agent: " +
agent.getName() + " on world: " + fromWorld.getName());
                        return false;
                  }
                  for (int i = 1; i < words.length; i++) {
                        if (words[i].trim().equalsIgnoreCase("X")) {
                              // -1 because the toWorld list is one shorter
than this line (it has no from).
                              state.addRelation(new Relation(agent,
fromWorld, toWorld[i - 1]));
                        }
                        else {
                              continue;
                        }
                  }
            }
            return true;
      }
}
```

## nl/rug/ai/believer/model/Core.java

```
package nl.rug.ai.believer.model;

/**
 * This class is the main (public) interface for any classes from other
packages to communicate
 * with the Believer package. This package will throw RuntimeExceptions in
case of illegal states
 * or unexpected problems, but should not raise any exceptions when used with
valid input.
 * It is both making use of the Singleton design pattern to ensure only one
instance of the program
 * runs at the same time, and the Facade pattern to provide access to this
program through one class.
 * @author S1849700
 *
 */
public class Core {

	private static Core core = null;
	private State state = new State();

	/**
	 * Singleton, private constructor
	 */
	private Core(){

	}

	/**
	 * Singleton, this method ensures there is at least (and at most) 1 core
and returns that.
	 * @return The requested instance.
	 */
	public static Core getCore() {
		if (core == null) {
			core = new Core();
		}
		return core;
	}

	/**
	 * This method reads a file, and loads all data from that file into the
core. (Clears all
	 * existing data first.)
	 * @param file the name of the file to read.
	 * @return true if the file was read successfully, false if not.
	 */
	public boolean readFile(String file) {
		BelieverFileReader reader = new BelieverFileReader();
		return reader.read(file, state);
	}

	/**
```

```
	 * This method determines if a formula is true or not in our current
model.
	 * @param statement The statement to be evaluated.
	 * @return The result of the evaluation.
	 */
	public boolean evaluate(String formula) {
		Evaluator eval = new Evaluator(state.clone());
		return eval.evaluate(formula);
	}

	/**
	 * This method generates a textual list of all worlds (1 name per line)
that exist in this core.
	 * @return The list of Worlds
	 */
	public String listWorlds() {
		return state.listWorlds();
	}

	/**
	 * This method creates a list of all relations that currently exist in
this core, per line there will
	 * be "agent name", "from world name" and "to world name".
	 * @return The list of Relations
	 */
	public String listRelations() {
		return state.listRelations();
	}

	/**
	 * This method makes a list of all agents that currently exist in this
core (1 name per line).
	 * @return The list of Agents
	 */
	public String listAgents() {
		return state.listAgents();
	}

	/**
	 * This method returns a list of all equivalence classes (listing agent
+ worlds for each class).
	 * @return The list of Equivalence Classes
	 */
	public String listEquivalenceClasses() {
		return state.listEquivalenceClasses();
	}

	/**
	 * This method allows you to set "verbose" to either true or false, if
true, debug statements may show.
	 * @param verbose the desired value of verbose.
	 */
	public void setVerbose(boolean verbose) {
		Util.verbose = verbose;
	}
```

```
	/**
	 * This method tells the Util class it should try to reload its config
file.
	 */
	public void reloadConfig() {
		Util.loadConfig();
	}
}
```

## nl/rug/ai/believer/model/EquivalenceClass.java

```
package nl.rug.ai.believer.model;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;

import nl.rug.ai.believer.model.exception.DuplicateException;

/**
 * A container class for equivalence classes. An equivalence class is a group
of worlds (or a single world) which can
 * not be distinguished by an agent on grounds of knowledge. (Within an
equivalence class, an agent could however
 * have preferences (beliefs) for one or more worlds.)
 *
 * This class does not make validity checks on agents/worlds, State is
responsible for keeping this class valid.
 * @author S1849700
 *
 */
public class EquivalenceClass {

	private Agent agent; // The agent to which this equiv-class belongs.
	/**
	 * Each element of this orderedBeliefs ArrayList contains an ArrayList
with Worlds that are (on a belief level)
	 * equivalent. The elements on the upper level (orderedBeliefs) are
sorted from most believed (element 0) to least
	 * believed (element size-1).
	 */
	private ArrayList<ArrayList<World>> orderedBeliefs = new
ArrayList<ArrayList<World>>();
	private ArrayList<World> knowledgeList = new ArrayList<World>();
	private ArrayList<Relation> relationList = new ArrayList<Relation>();

	EquivalenceClass(Agent agent) {
		this.agent = agent;
	}

	/**
	 * Add a world to this equivalence class. (All worlds of an equivalence
class are on a knowledge level equivalent
	 * to an agent.
	 * @param world the world to add
	 */
	private void addWorld(World world) {
		if (hasWorld(world)) {
			return; // we already have this world, no need for
duplicates.
		}
		this.knowledgeList.add(world);
	}
```

```
	/**
	 * Add a relation to this equivalence Class. By adding a relation,
knowledge and belief of this equiv class are
	 * automatically updated.
	 * @param relation the relation to add
	 */
	void addRelation(Relation relation) {
		if (relationList.contains(relation)) {
			throw new DuplicateException("Duplicate relation for
agent: " + relation.getAgent().getName() +
					" from world: " +
relation.getFromWorld().getName() + " to world: " +
					relation.getToWorld().getName());
		}
		if (!relation.getAgent().equals(agent)) {
			throw new IllegalArgumentException("A relation of the
wrong agent was added to an equivalence class.");
		}
		relationList.add(relation);
		addWorld(relation.getFromWorld());
		addWorld(relation.getToWorld());
		updateBeliefs();
	}

	/**
	 * This method creates a new list of believed worlds.
	 */
	private void updateBeliefs() {
		Integer occurances[] = getWorldOccurance();
		Integer sortedOccurances[] = occurances.clone();
		Arrays.sort(sortedOccurances, Collections.reverseOrder()); //
sort from high to low.
		ArrayList<ArrayList<World>> orderedBList = new
ArrayList<ArrayList<World>>();
		int lastSeen = -1;
		for (int current : sortedOccurances) {
			if (current == lastSeen) {
				continue;
			}
			ArrayList<World> wList = new ArrayList<World>();
			for (int i = 0; i < occurances.length; i++) {
				if (occurances[i] == current) {
					wList.add(knowledgeList.get(i));
				}
			}
			orderedBList.add(wList);
			lastSeen = current;
		}
		orderedBeliefs = orderedBList;
	}

	/**
	 * Determine how many relations in this equiv class have each world as
their "toWorld". Sorted in the same way
	 * as knowledgeList.
```

```
	 * @return an array containing this information (index matches the
knowledgeList ArrayList).
	 */
	private Integer[] getWorldOccurance() {
		Integer result[] = new Integer[knowledgeList.size()];
		for (int i = 0; i < result.length; i++) {
			result[i] = new Integer(0);
		}
		for (Relation rel : relationList) {
			int nr = knowledgeList.indexOf((rel.getToWorld()));
			if (nr == -1) {
				throw new IllegalStateException("An instance
Equivalence Class has become corrupt and cannot continue" +
					" to operate properly.");
			}
			result[nr]++;
		}
		return result;
	}

	/**
	 * Check if a world exists in this equivalence class.
	 * @param world the world to check
	 * @return true if exists
	 */
	boolean hasWorld(World world) {
		return knowledgeList.contains(world);
	}

	/**
	 * Returns the agent object associated with this equivalence class
	 * @return the agent object to who this class belongs
	 */
	Agent getAgent() {
		return agent;
	}

	/**
	 * This method returns a list of worlds that constitutes the "knowledge"
in this equivalence class.
	 * @return array of worlds
	 */
	World[] getKnowledge() {
		World[] myWorlds = new World[knowledgeList.size()];
		knowledgeList.toArray(myWorlds);
		return myWorlds;
	}

	/**
	 * This method returns a list of worlds that constitutes the "belief" in
this equivalence class.
	 * @return array of worlds
	 */
	World[] getBelief() {
		if (orderedBeliefs.isEmpty()) {
```

```
                return new World[0];
            }
            World[] myWorlds = new World[orderedBeliefs.get(0).size()];
            orderedBeliefs.get(0).toArray(myWorlds);
            return myWorlds;
        }

        /**
         * This method returns the list of relations which form this equivalence
class.
         * @return array of worlds
         */
        Relation[] getRelations() {
            Relation[] myRelations = new Relation[relationList.size()];
            relationList.toArray(myRelations);
            return myRelations;
        }

        /**
         * this method returns an ArrayList of ArrayLists containing worlds,
which represents the full
         * ordered belief as present in this equivalence class.
         * @return the ordered belief
         */
        ArrayList<ArrayList<World>> getOrderedBeliefs() {
            return orderedBeliefs;
        }
}
```

## nl/rug/ai/believer/model/Evaluator.java

```
package nl.rug.ai.believer.model;

import java.util.ArrayList;

/**
 * This class is responsible for doing any actual evaluations. For this
purpose, the core will provide it
 * with a copy of current state.
 * @author S1849700
 *
 */
public class Evaluator {

	private State state;

	Evaluator(State state) {
		this.state = state;
	}

	/**
	 * external interface for the rest of the package. This method splits
the formula on the turnstyle and then calls
	 * eval() to do the rest. (calls it on all worlds if there is no world
before the turnstyle.
	 * @param formula  the formula to evaluate.
	 * @return the result of the evaluation.
	 */
	boolean evaluate(String formula) {
		int turnstylePos = formula.indexOf(Util.turnstyle);
		if (turnstylePos != -1) {
			if (formula.substring(0, turnstylePos).trim().equals(""))
{
				// if there is nothing before the turnstyle, by
convention, we want to evaluate on all worlds.
				String myFormula = formula.substring(turnstylePos +
Util.turnstyle.length()).trim();
				for (World world : state.getWorlds()) {
					if(!eval(myFormula, world)) { // must be true
on all worlds.
						return false;
					}
				}
				// if it didn't return before this point, it indeed
is true.
				// incidentally, this causes all formula's to be
true which are made when there are no worlds at all...
				return true;
			}
			else {
				World world = state.getWorld(formula.substring(0,
turnstylePos).trim());
				if (world == null) { // can't eval, origin world
does not exist
```

```
                                    throw new RuntimeException("Cannot evaluate,
world before the turnstyle does not exist.");
                              }
                              return eval(formula.substring(turnstylePos +
Util.turnstyle.length()).trim(), world);
                        }
                  }
                  throw new RuntimeException("Cannot evaluate, a formula without
turnstyle is not a valid formula.");
            }


            /**
             * this method calls itself recursively (if needed) to solve  the
formula.
             * @param formula the formula to solve
             * @param world  the world in which to solve it.
             * @return the result of the evaluation
             */
            private boolean eval(String formula, World world) {
                  if (formula.trim().isEmpty()) {
                        Util.debug("Empty formula.");
                        return false;
                  }
                  String myFormula = removeOuterParenthesis(formula);
                  while (!myFormula.equals(removeOuterParenthesis(myFormula))) {
                        myFormula = removeOuterParenthesis(myFormula);  // remove
arbitrary amount of parenthesis.
                  }
                  String formulaParts[];
                  formulaParts = splitOperator(Util.equals, myFormula);
                  if (formulaParts != null) { // true if both sides are equal
                        Util.debug("Chose the equals operator in: " + myFormula);
                        return eval(formulaParts[0], world) ==
eval(formulaParts[1], world);
                  }
                  formulaParts = splitOperator(Util.implies, myFormula);
                  if (formulaParts != null) { // true if first part does not hold,
or if first and second do.
                        Util.debug("Chose the implies operator in: " + myFormula);
                        if (eval(formulaParts[0], world)) {
                              return eval(formulaParts[1], world);
                        }
                        else {
                              return true;
                        }
                  }
                  formulaParts = splitOperator(Util.or, myFormula);
                  if (formulaParts != null) { // true if either side of the OR is
true.
                        Util.debug("Chose the or operator in: " + myFormula);
                        return eval(formulaParts[0], world) ||
eval(formulaParts[1], world);
                  }
                  formulaParts = splitOperator(Util.and, myFormula);
```

```
		if (formulaParts != null) { // true only if both sides are true
			Util.debug("Chose the and operator in: " + myFormula);
			return eval(formulaParts[0], world) &&
eval(formulaParts[1], world);
		}

		String unary = obtainUnaryOperator(Util.not, myFormula);
		if (unary != null) { // NOT operator just reverses the outcome.
			Util.debug("Chose the not operator in: " + myFormula);
			return !eval(unary, world);
		}

		unary = obtainUnaryOperator(Util.belief, myFormula);
		if (unary != null) {
			Util.debug("Chose the belief operator in: " + myFormula);
			Agent agent = extractAgent(myFormula);
			if (agent == null) {
				Util.debug("Invalid agent in: " + myFormula);
				throw new RuntimeException("Invalid agent"); //
invalid agent, can't evaluate.
			}
			formulaParts = splitComplexUnaryOperator(Util.conditional,
unary);
			if (formulaParts == null) { // regular belief
				for(World myWorld : state.getBeliefs(agent, world))
{
					if (!eval(unary, myWorld)) {
						return false;
					}
				}
				return true;
			}
			// conditional belief
			ArrayList<ArrayList<World>> orderedBeliefs =
state.getOrderedBeliefs(agent, world);
			if (orderedBeliefs == null || orderedBeliefs.size() == 0)
{
				throw new RuntimeException("Program entered illegal
state, can't continue evaluations.");
			}
			boolean found = false;
			for (ArrayList<World> set : orderedBeliefs) {
				for (World myWorld : set) {
					// found a world where the condition holds
					if (eval(formulaParts[0], myWorld)) {
						// if the rest of the formula does not
hold on *any* world where the condition does, the
						// total result is false.
						if(!eval(formulaParts[1], myWorld)) {
							return false;
						}
						found = true;
					}
				}
```

```
                        // means one of the ordered groups of worlds
contained one or more worlds where this was true.
                        // we do not want to search lower-believed worlds in
that case.
                        if (found) {
                              break;
                        }
                  }
                  if (!found) {
                        Util.debug("No world match conditional, defaulting
to false.");
                        return false;
                  }
                  // If we are here, then one or more worlds were found
where the conditional holds, and the rest as well.
                  return true;
            }
            unary = obtainUnaryOperator(Util.knowledge, myFormula);
            if (unary != null) {
                  Util.debug("Chose the knowledge operator in: " +
myFormula);
                  Agent agent = extractAgent(myFormula);
                  if (agent == null) {
                        Util.debug("Invalid agent in: " + myFormula);
                        throw new RuntimeException("Invalid agent"); //
invalid agent, can't evaluate.
                  }
                  // get all worlds known to this agent, if even 1 of them
is false, it is not knowledge, otherwise true.
                  for(World myWorld : state.getKnowledgeWorlds(agent,
world)) {
                        if (!eval(unary, myWorld)) {
                              return false;
                        }
                  }
                  return true;
            }
            formulaParts = splitComplexUnaryOperator(Util.update, myFormula);
            if (formulaParts != null) {
                  Util.debug("Chose the update operator in: " + myFormula);

                  // search for agents
                  ArrayList<Agent> agents =
extractAgents(myFormula.substring(Util.update.length()));
                  // make a copy of the state, and use a new clone of it, as
we will temporarily want to change some things.
                  State old = state;
                  state = state.clone();

                  // collect worlds which are true according to our update
condition.
                  ArrayList<World> targetWorlds = new ArrayList<World>();
                  for (World target : state.getWorlds()) {
                        if (eval(formulaParts[0], target)) {
                              targetWorlds.add(target);
```

```
                        }
                }

                World myWorld = world;
                // if no agents were found, this command was meant for
all. (special case)
                if (agents == null) {
                        state.performUpdate(targetWorlds);
                }
                else {
                        myWorld = state.performUpdate(targetWorlds, agents,
world);
                }

                // Evaluate if after the update, the formula holds.
                boolean result = eval(formulaParts[1], myWorld);

                state = old; // return state to normal
                return result;
            }
            formulaParts = splitComplexUnaryOperator(Util.radical_upgrade,
myFormula);
            if (formulaParts != null) {
                Util.debug("Chose the radical upgrade operator in: " +
myFormula);

                // search for agents
                ArrayList<Agent> agents =
extractAgents(myFormula.substring(Util.radical_upgrade.length()));
                // make a copy of the state, and use a new clone of it, as
we will temporarily want to change some things.
                State old = state;
                state = state.clone();

                // determine worlds matching the upgrade condition
                ArrayList<World> targetWorlds = new ArrayList<World>();
                for (World target : state.getWorlds()) {
                        if (eval(formulaParts[0], target)) {
                                targetWorlds.add(target);
                        }
                }

                World myWorld = world;
                // if no agents were found, this command was meant for
all. (special case)
                if (agents == null) {
                        state.performUpgrade(targetWorlds);
                }
                else {
                        myWorld = state.performRadicalUpgrade(targetWorlds,
agents, world);
                }

                // Evaluate if after the upgrade, the formula holds.
                boolean result = eval(formulaParts[1], myWorld);
```

```
                        state = old; // return state to normal
                        return result;
                }
                formulaParts =
splitComplexUnaryOperator(Util.conservative_upgrade, myFormula);
                if (formulaParts != null) {
                        Util.debug("Chose the conservative upgrade operator in: "
+ myFormula);

                        // make a copy of the state, and use a new clone of it, as
we will temporarily want to change some things.
                        State old = state;
                        state = state.clone();
                        ArrayList<World> targetWorlds = new ArrayList<World>();

                        for (Agent agent : state.getAgents()) {
                                ArrayList<ArrayList<World>> orderedBeliefs =
state.getOrderedBeliefs(agent, world);
                                if (orderedBeliefs == null || orderedBeliefs.size()
== 0) {
                                        throw new RuntimeException("Program entered
illegal state, can't continue evaluations.");
                                }
                                boolean found = false;
                                for (ArrayList<World> set : orderedBeliefs) {
                                        for (World myWorld : set) {
                                                // found a world where the condition
holds
                                                if (eval(formulaParts[0], myWorld)) {
                                                        targetWorlds.add(myWorld);
                                                        found = true;
                                                }
                                        }
                                        // means one of the ordered groups of worlds
contained one or more worlds where this was true.
                                        // we do not want to search lower-believed
worlds in that case.
                                        if (found) {
                                                break;
                                        }
                                }
                                state.performConvervativeUpgrade(targetWorlds,
agent);
                                targetWorlds.clear();
                        }

                        // Evaluate if after the upgrade, the formula holds.
                        boolean result = eval(formulaParts[1], world);

                        state = old; // return state to normal
                        return result;
                }
```

```
            // if we get here, then no operators were found, thus we (should)
have a single proposition left.
            if (world.containsProposition(myFormula)) {
                  Util.debug("proposition " + myFormula + " was true on
world: " + world.getName());
                  return true;
            }
            Util.debug("proposition " + myFormula + " was false on world: " +
world.getName());
            return false;

      }

      /**
       * Obtain the agent from a formula. This should only be called on belief
or knowledge operators. Missing open or close
       * brackets will lead to an unknown agent. Since it can only contain an
agent name, the start and end tags matched are
       * the first found in the String. (note that the start tag must directly
follow the B/K symbol.
       * @param formula the formula to inspect
       * @return the resulting agent. (or null if not found)
       */
      private Agent extractAgent(String formula) {
            int start = formula.indexOf(Util.agentStartTag);
            int end = formula.indexOf(Util.agentEndTag);
            if (start == -1 || end == -1 || start > 1) { // more than 1 means
it's not following this B or K
                  return null;
            }
            return state.getAgent(formula.substring(start + 1, end).trim());
      }

      /**
       * This method returns an ArrayList containing all agents found within
the agent tags.
       * Agents separated by the agentSeparator (as defined in the config).
Throws a RuntimeException
       * if any agents are invalid or not existent.
       * @param formula The formula at the start of which the agent tags are
expected.
       * @return an ArrayList with the agents found, or null if no
agentStartTag/end tag were found.
       */
      private ArrayList<Agent> extractAgents(String formula) {
            int start = formula.indexOf(Util.agentStartTag);
            int end = formula.indexOf(Util.agentEndTag);
            if (start == -1 || end == -1 || start > 1) { // more than 1 means
it's not following this B or K
                  return null;
            }
            ArrayList<Agent> result = new ArrayList<Agent>();
            String[] agents = formula.substring(start + 1,
end).trim().split(Util.agentSeparator);
            if (agents.length == 0) {
```

```
                        throw new RuntimeException("Found agent start/end tag
without any agent.");
                }
                for (String agent : agents) {
                        if (state.getAgent(agent.trim()) != null) {
                                result.add(state.getAgent(agent.trim()));
                        }
                        else {
                                throw new RuntimeException("Invalid/Unknown agent
name: " + agent);
                        }
                }
                return result;
        }

        /**
         * This method removes the outer parenthesis around a string (if any)
and returns the cleaned
         * string, or the original of none were found.
         * @param input -- the input to be cleaned
         * @return -- the cleaned input
         */
        private String removeOuterParenthesis(String input) {
                String myInput = input.trim();
                if (myInput.equals("")) {
                        return myInput;
                }
                if (myInput.charAt(0) == '(' && myInput.charAt(myInput.length() -
1) == ')') {
                        int depth = 0;
               // -1 because we want to exclude the last ) from the check, after
all,that one would certainly make the depth 0.
               for(int i = 0; i < myInput.length() - 1; i++) {
                    if (myInput.charAt(i) == '(') {
                        depth++;
                    }
                    else if (myInput.charAt(i) == ')') {
                        depth--;
                    }
                    // This means the first ( gets closed somewhere (and thus
there is (should be) an operator we need first.
                    if (depth <= 0) {
                        return myInput;
                    }
               }
               // -1 because it takes endIndex-1, thus this excludes the )
               return myInput.substring(1, myInput.length() - 1).trim();
         }
                return myInput;
        }

        /**
         * Splits the formula on the operator (if found). Returns an array
containing the formula on the left in position 0
```

```
        * and the right in position 1.  returns null if the operator was not
found.
        * @param operator  The operator to split on
        * @param formula  the formula to search in
        * @return an array containing the left and right side formula's of the
operator. Or null if the operator was not found.
        */
       private String[] splitOperator(String operator, String formula) {
             String myInput = formula.trim();
             int depth = 0;
             String result[] = new String[2];
        for(int i = 0; i < myInput.length(); i++) {
             // In this loop, we ignore all blocks that are between ()
            if (myInput.charAt(i) == '(') {
                depth++;
            }
            else if (myInput.charAt(i) == ')') {
                depth--;
            }
            // This means the first ( gets closed somewhere (and thus there is
(should be) an operator we need first.
            if (depth != 0) {
                continue;
            }
            // means we found the operator we are looking for at this spot
(i).
            if (myInput.startsWith(operator, i)) {
             result[0] = myInput.substring(0, i).trim();
             result[1] = myInput.substring(i + operator.length()); // start at
the first char after this operator.
             return result;
            }
        }
             // if we reach this, we didn't find our operator.
             return null;
       }

       /**
        * Method returns all that follows the unary operator (not the operator
itself). <agent> is considered part of the
        * operator. Returns null if not found.
        * @param operator the operator to look for
        * @param formula the formula in which to look for the operator.
        * @return the formula following the operator or null if not found.
        */
       private String obtainUnaryOperator(String operator, String formula) {
             String myInput = formula.trim();
             if (operator.equals(Util.not)) {
                   if (myInput.startsWith(operator)) {
                         return myInput.substring(operator.length());
                   }
                   return null;
             }
             // agentStart and End tag always follow each other (no formulas
between them, just an agent name.
```

```
            // We assume that agentStartTag follows directly after a B/K
symbol.
            if (myInput.startsWith(operator + Util.agentStartTag)) {
                  if (myInput.indexOf(Util.agentEndTag) == -1) {
                        throw new RuntimeException("Found an agentStartTag
without matching agentEndTag");
                  }
                  return myInput.substring(myInput.indexOf(Util.agentEndTag)
+ 1); // +1 to get all that is *after* the tag.
            }
            return null;
      }

      /**
       * used to split complex unary operators (update, upgrade, conditional,
etc) like UPDATE(formula1)formula2.
       * It will return formula 1 at position 0 and formula 2 and position 1
of the array.
       * @return
       */
      private String[] splitComplexUnaryOperator(String operator, String
formula) {
            String myInput = formula.trim();
            String result[] = new String[2];
            if (formula == null || formula.trim().equals("")) {
                  return null;
            }
            if (myInput.startsWith(operator)) {
                  myInput = myInput.substring(operator.length()).trim();
                  ArrayList<Agent> temp = extractAgents(myInput);
                  if (temp != null) {
                        myInput =
myInput.substring(myInput.indexOf(Util.agentEndTag) +
Util.agentEndTag.length());
                  }
                  if (!myInput.startsWith("(")) {
                              return null;
                  }
                  // we start at 1, because our first char (0) is the first
opening.
                  int depth = 1;
              for(int i = 1; i < myInput.length(); i++) {
                  if (myInput.charAt(i) == '(') {
                      depth++;
                  }
                  else if (myInput.charAt(i) == ')') {
                      depth--;
                  }
                  // This means we found the closing of this block.
                  if (depth == 0) {
                   // 1 is the position just past the opening, substring uses
end-1, thus excludes the closing. (this i)
                      result[0] = myInput.substring(1, i).trim();
                      // and +1 to get all that follows the () block.
                      result[1] = myInput.substring(i + 1).trim();
```

```
                    return result;
                }
            }
        }
        return null;
    }
}
```

## nl/rug/ai/believer/model/Relation.java

```
package nl.rug.ai.believer.model;

/**
 * This class represents a relation between 2 worlds. Each relation is owned
by an agent.
 *
 * @author S1849700
 *
 */
public class Relation {

	private Agent agent;
	private World from;
	private World to;

	/**
	 * Standard constructor, used to initialize the agent, the from world
and the to world of this relation.
	 * @param agent the agent to who this relation belongs
	 * @param from the from world from which this relation originates.
	 * @param to the to world, to where this relation goes. (In case of
refelxive relations, identical to from.
	 */
	public Relation(Agent agent, World from, World to) {
		if (agent == null || from == null || to == null) {
			throw new NullPointerException("Contructor element of
Relation was null.");
		}
		this.agent = agent;
		this.from = from;
		this.to = to;
	}

	/**
	 * accessor method for the agent of this relation
	 * @return the agent.
	 */
	public Agent getAgent() {
		return agent;
	}

	/**
	 * accessor method for the to world
	 * @return the to world
	 */
	public World getToWorld() {
		return to;
	}

	/**
	 * accessor method for the from world
	 * @return the from world
	 */
```

```
	public World getFromWorld() {
		return from;
	}

	/**
	 * We consider an Agent equal if their name is equal.
	 * @see Object#equals(Object)
	 */
	@Override
	public boolean equals(Object other) {
	    //check for self-comparison
	    if ( this == other ) {
	      return true;
	    }
	    // this renders an explicit check for "other == null" redundant,
since it does the check for
	    // null already - "null instanceof [type]" always returns false.
	    if (!(other instanceof Relation)) {
	      return false;
	    }
	    Relation otherRelation = (Relation)other;
	    // if agent, from and to worlds are equal, the relation is equal.
	    return otherRelation.agent.equals(agent) &&
otherRelation.from.equals(from) && otherRelation.to.equals(to);
	}

	/**
	 * @see Object#hashCode(Object)
	 */
	// must be overwritten if .equals is overwritten. (And should be based
on the same key fields.)
	@Override
	public int hashCode() {
		return (agent.hashCode() * 31 + from.hashCode()) * 31 +
to.hashCode();
	}

}
```

## nl/rug/ai/believer/model/State.java

```
package nl.rug.ai.believer.model;

import java.util.ArrayList;

import nl.rug.ai.believer.model.exception.DuplicateException;
import nl.rug.ai.believer.model.exception.InvalidAgentException;
import nl.rug.ai.believer.model.exception.InvalidWorldException;

/**
 * This class represents a complete state. It contains a list of agents,
worlds and relations,
 * as well a full set of equivalence classes. Finally, it is responsible for
printing its contents
 * and for performing updates and upgrades.
 * @author S1849700
 *
 */
public class State implements Cloneable {

	private ArrayList<Agent> agentList = new ArrayList<Agent>();
	private ArrayList<World> worldList = new ArrayList<World>();

	// This is technically duplicated over all equiv classes, but useful for
updates/upgrades to reconstruct them.
	private ArrayList<Relation> relationList = new ArrayList<Relation>();
	private ArrayList<EquivalenceClass> equivalenceClassList = new
ArrayList<EquivalenceClass>();

	/**
	 * This method can be used to copy another State into this one. It is
assumed the "other" State
	 * does not change the equiv class list anymore. Otherwise this one will
change with it.
	 * @param other another state, to be copied into this one.
	 */
	void copyFromOther(State other) {
		agentList = new ArrayList<Agent>(other.agentList);
		worldList = new ArrayList<World>(other.worldList);
		relationList = new ArrayList<Relation>(other.relationList);
		equivalenceClassList = new
ArrayList<EquivalenceClass>(other.equivalenceClassList);
	}

	/**
	 * returns all worlds from the same equivalence class as the provided
world. (Knowledge)
	 * @param agent Agent for which we want knowledge.
	 * @param world The world which will be used as origin.
	 * @return All worlds existing in the equivalence class that matches
this agent/world.
	 */
	World[] getKnowledgeWorlds(Agent agent, World world) {
		for (EquivalenceClass ec : equivalenceClassList) {
```

```
                if (ec.getAgent().equals(agent) && ec.hasWorld(world)) {
                        return ec.getKnowledge();
                }
        }
        return null;
    }

    /**
     * Returns an ordered belief construct as contained in the equivalence
class. The ordered belief
     * returned belongs to a specific agent on a specific world.
     * @param agent - the agent for who to retrieve the ordered belief
     * @param world - the world for which the ordered belief needs to be
found
     * @return  - the ordered belief
     */
    ArrayList<ArrayList<World>> getOrderedBeliefs (Agent agent, World world)
{
        for (EquivalenceClass ec : equivalenceClassList) {
                if (ec.getAgent().equals(agent) && ec.hasWorld(world)) {
                        return ec.getOrderedBeliefs();
                }
        }
        return null;
    }

    /**
     * This method returns an array of worlds which are believed on a
specific world for a
     * specific agents.
     * @param agent - the agent for which to find the beliefs
     * @param world - the world for which to find the beliefs
     * @return - the resulting believed worlds in array form.
     */
    World[] getBeliefs (Agent agent, World world) {
        for (EquivalenceClass ec : equivalenceClassList) {
                if (ec.getAgent().equals(agent) && ec.hasWorld(world)) {
                        return ec.getBelief();
                }
        }
        return null;
    }

    /**
     * returns an agent with a certain name (if found) or null (if not).
(looks in the internal list
     * of this object)
     * @param agent The name of the agent we are looking for
     * @return The result
     */
    Agent getAgent(String agent) {
        int result = agentList.indexOf(new Agent(agent));
        return (result == -1 ? null : agentList.get(result));
    }
```

```
	/**
	 * returns an ArrayList containing all agents known to this state.
	 * @return the list of agents
	 */
	ArrayList<Agent> getAgents() {
		return new ArrayList<Agent>(agentList);
	}

	/**
	 * returns a world with a certain name (if found), null otherwise (looks
in the internal list
	 * of this object)
	 * @param world the desired world name
	 * @return the result
	 */
	World getWorld(String world) {
		int result = worldList.indexOf(new World(world));
		return (result == -1 ? null : worldList.get(result));
	}

	/**
	 * Returns an ArrayList containing all worlds known to this State.
	 * @return the list of worlds
	 */
	ArrayList<World> getWorlds() {
		return new ArrayList<World>(worldList);
	}

	/**
	 * Check if a world exists in this state
	 * @param world world to be checked
	 * @return result true if found, false if not.
	 */
	boolean hasWorld(World world) {
		return worldList.contains(world);
	}
		/**
	 * This method generates a textual list of all worlds (1 name per line)
that exist in this state.
	 * @return The list of Worlds
	 */
	String listWorlds() {
		StringBuffer sb = new StringBuffer();
		sb.append("Listing all world names: \n");
		for (World world : worldList) {
			sb.append(world.getName() + "\n");
		}
		sb.append("\n");
		return sb.toString();
	}

	/**
	 * This method creates a list of all relations that currently exist in
this state, per line there will
	 * be "agent name", "from world name" and "to world name".
```

```
	 * @return The list of Relations
	 */
	String listRelations() {
		StringBuffer sb = new StringBuffer();
		sb.append("Listing all relations as <agent>, <from world>, <to
world>: \n");
		for (Relation relation : relationList) {
			sb.append(relation.getAgent().getName() + ", " +
relation.getFromWorld().getName() + ", " +
					relation.getToWorld().getName() + "\n");
		}
		return sb.toString();
	}

	/**
	 * This method makes a list of all agents that currently exist in this
state (1 name per line).
	 * @return The list of Agents
	 */
	String listAgents() {
		StringBuffer sb = new StringBuffer();
		sb.append("Listing all agent names: \n");
		for (Agent agent : agentList) {
			sb.append(agent.getName() + "\n");
		}
		sb.append("\n");
		return sb.toString();
	}

	/**
	 * This method makes a list of all equivalence classes which currently
exist in this state.
	 * format is line 1: number, line 2: agent name, line 3: worlds, and
then repeat for each
	 * equivalence class.
	 * @return - the list of equivalence classes
	 */
	String listEquivalenceClasses() {
		StringBuffer sb = new StringBuffer();
		sb.append("Listing all equivalence classes");
		for (int i = 0; i < equivalenceClassList.size(); i++) {
			sb.append("\n\nEquivalence Class nr. " + i + "\n");
			sb.append("Agent Name: " +
equivalenceClassList.get(i).getAgent().getName() + "\n");
			sb.append("Worlds: ");
			for (World w : equivalenceClassList.get(i).getKnowledge())
{
				sb.append(w.getName() + ", ");
			}
			sb.deleteCharAt(sb.length() - 2); // delete the comma at
the end of the last item
		}
		return sb.toString();
	}
```

```
	/**
	 * Method to add an agent to this state
	 * @param agent - the agent to be added
	 */
	void addAgent(Agent agent) {
		if (agentList.indexOf(agent) != -1) {
			throw new DuplicateException("Duplicate agent: " +
agent.getName());
		}
		agentList.add(agent);
	}

	/**
	 * Method to add a world to this state
	 * @param world - the world to be added
	 */
	void addWorld(World world) {
		if (worldList.indexOf(world) != -1) {
			throw new DuplicateException("Duplicate world: " +
world.getName());
		}
		worldList.add(world);
	}

	/**
	 * Method to add a relation to this state
	 * @param relation - the relation to be added
	 */
	void addRelation(Relation relation) {
		if (relationList.contains(relation)) {
			throw new DuplicateException("Duplicate relation, agent: "
+ relation.getAgent().getName()
					+ " from: " +
relation.getFromWorld().getName() + " to: " +
					relation.getToWorld().getName());
		}
		if (!agentList.contains(relation.getAgent())) {
			throw new InvalidAgentException("Cannot add a relation for
which no agent exists ("
					+ relation.getAgent().getName() + ")");
		}
		if (!worldList.contains(relation.getFromWorld())) {
			throw new InvalidWorldException("Cannot add a relation for
which the 'from world' does not exist ("
					+ relation.getFromWorld().getName() + ")");
		}
		if (!worldList.contains(relation.getToWorld())) {
			throw new InvalidWorldException("Cannot add a relation for
which the 'to world' does not exist ("
					+ relation.getToWorld().getName() + ")");
		}
		relationList.add(relation);

		addToEquivalenceClass(relation);
	}
```

```
	/**
	 * Support method to add a relation to the proper equivalence class.
Also responsible for merging
	 * equivalence classes if need be.
	 * @param relation - the relation to be added
	 */
	private void addToEquivalenceClass(Relation relation) {
		EquivalenceClass existsIn1 = null;
		EquivalenceClass existsIn2 = null;

		for (EquivalenceClass ec : equivalenceClassList) {
			if (!ec.getAgent().equals(relation.getAgent())) {
				continue; // not the same agent means we're not
interested.
			}
			// EquivalenceClass will catch any duplicates that arise
from this.
			if (ec.hasWorld(relation.getFromWorld()) ||
ec.hasWorld(relation.getToWorld())) {
				if (existsIn1 == null) {
					existsIn1 = ec;
				}
				else {
					if (existsIn2 != null) {
						throw new RuntimeException("Found 3
matching Equivalence Classes to 1 relation, 2 is max.");
					}
					existsIn2 = ec;
				}
			}
		}
		// Adding a relation to an equiv class.
		if (existsIn1 != null && existsIn2 == null) {
			existsIn1.addRelation(relation);
		}
		// merging 2 equiv classes if needed
		else if (existsIn1 != null && existsIn2 != null) {
			Util.debug("Found worlds in this relation referenced in 2
equivalence classes, merging them.");
			Util.debug("Relation belongs to agent " +
relation.getAgent().getName() + " from world " +
					relation.getFromWorld().getName() + " to
world " + relation.getToWorld().getName());

			existsIn1.addRelation(relation);
			for (Relation rel : existsIn2.getRelations()) {
				existsIn1.addRelation(rel);
			}
			equivalenceClassList.remove(existsIn2);
		}
		// Creating a new equiv class for this relation (these worlds)
		else {
			EquivalenceClass ec = new
EquivalenceClass(relation.getAgent());
```

```
                        ec.addRelation(relation);
                        equivalenceClassList.add(ec);
                }
        }

        /**
         * This method makes an update based on global announcement policies (no
agents/private knowledge involved).
         * Also note that this *changes* the state by deleting relations and the
like, so usage of a clone is recommended.
         * @param validWorlds the list of worlds that is "valid" and needs to be
separated from the rest.
         */
        void performUpdate(ArrayList<World> validWorlds) {
                // every delete makes this list shorter so work from back to
front (or you'll need to mess with the counter)
                for (int i = relationList.size() - 1; i != 0; i--) {
                        Relation target = relationList.get(i);
                        // if a relation contains a valid toworld but not a valid
from world (or vice versa), it needs to be deleted.
                        if ( ( validWorlds.contains(target.getFromWorld()) &&
!validWorlds.contains(target.getToWorld()) ) ||
                                        (
!validWorlds.contains(target.getFromWorld()) &&
validWorlds.contains(target.getToWorld()) ) ) {
                                relationList.remove(i);
                        }
                }
                // clear all equiv classes
                equivalenceClassList = new ArrayList<EquivalenceClass>();
                // and refill them.
                for (Relation rel : relationList) {
                        addToEquivalenceClass(rel);
                }
        }

        /**
         * This method performs a secret update for the agents listed in
"agents", in which all "validWorlds"
         * are assumed to be true. Furthermore, it will return an updated
reference to the "currentWorld"
         * which will point at the newly created world. (Rather than the old
one.)
         * @param validWorlds - all worlds on which the condition of the update
is true
         * @param agents - all agents who are aware of this update
         * @param currentWorld - the current world
         * @return -- an updated reference to the current world
         */
        World performUpdate(ArrayList<World> validWorlds, ArrayList<Agent>
agents, World currentWorld) {
                ArrayList<World> newWorldList = new ArrayList<World>();
                ArrayList<Relation> relList = new ArrayList<Relation>();
                World newCurrentWorld = currentWorld;
                for (World validWorld : validWorlds) {
```

```
                    for (long i = 0; ; i++) {
                          // if we exceed this value, we can't continue.
Should not happen in normal operation
                          // without extreme computing power, or it will take
literally years to calculate.
                          if (i >= Long.MAX_VALUE - 1) {
                                throw new RuntimeException("Error: too much
duplicates of World \"" +
                                      validWorld.getName() + "\". Under
normal conditions, this error can only be "
                                      + "reached by formula's which will take
close to forever (years) to validate");
                          }
                          World myWorld = new World(validWorld);
                          // long + string = string in the form of 1234string
                          myWorld.setName(i + myWorld.getName());
                          // get the next number if this one already exists.
                          if (worldList.contains(myWorld)) {
                                continue;
                          }
                          worldList.add(myWorld);
                          newWorldList.add(myWorld);
                          if (validWorld.equals(currentWorld)) {
                                newCurrentWorld = myWorld;
                          }
                          for (Agent myAgent : agentList) {
                                // add reflexive relations for all agents.
                                relList.add(new Relation(myAgent, myWorld,
myWorld));
                                // if agent is not in the list, he believes
the old world is true.
                                if (!agents.contains(myAgent)) {
                                      relList.add(new Relation(myAgent,
myWorld, validWorld));
                                }
                          }
                          break; // if we are here, we are done for with this
world. (no more numbers needed.)
                    }
              }
              for (Relation target : relationList) {
                    // if both parts of the relation are within the valid
worlds, and the relation is not reflexive
                    if ( validWorlds.contains(target.getFromWorld()) &&
validWorlds.contains(target.getToWorld())
                                &&
!target.getFromWorld().equals(target.getToWorld())) {
                          // find which new worlds this relation was about
                          World from =
newWorldList.get(validWorlds.indexOf(target.getFromWorld()));
                          World to =
newWorldList.get(validWorlds.indexOf(target.getToWorld()));
                          // add it to a temp list (no need to mess with the
iterator of this loop).
```

```
                        relList.add(new Relation(target.getAgent(), from,
to));
                  }
                  // if a relation contains a valid fromworld but not a
valid toworld, and agent is not listed
                  if (validWorlds.contains(target.getFromWorld()) &&
!validWorlds.contains(target.getToWorld())
                              && !agents.contains(target.getAgent())) {
                        // find which new world this relation was from
                        World from =
newWorldList.get(validWorlds.indexOf(target.getFromWorld()));
                        // add relation from new world to old for this agent
                        Relation rel = new Relation(target.getAgent(), from,
target.getToWorld());
                        // may be duplicate, see next if block.
                        if (!relList.contains(rel)) {
                              relList.add(rel);
                        }
                  }
                  // if agent not listed *and* relation goes from outside to
inside, there must be an arrow from inside to out. (connected)
                  if (validWorlds.contains(target.getToWorld()) &&
!validWorlds.contains(target.getFromWorld())
                              && !agents.contains(target.getAgent())) {
                              // get new version of ToWorld, make it the
new from. These worlds may be duplicates.
                              World from =
newWorldList.get(validWorlds.indexOf(target.getToWorld()));
                              Relation rel = new
Relation(target.getAgent(), from, target.getFromWorld());
                              if (!relList.contains(rel)) {
                                    relList.add(rel);
                              }
                  }
                  // in all other cases (reflexive and from old to new and
from new to old for agents that
                  // were in the list, do nothing. (Reflexive is already
added earlier for all agents.)
            }
            relationList.addAll(relList);

            // clear all equiv classes
            equivalenceClassList = new ArrayList<EquivalenceClass>();
            // and refill them.

            for (Relation rel : relationList) {
                  addToEquivalenceClass(rel);
            }
            return newCurrentWorld;
      }

      /**
       * This method performs a secret radical upgrade for the agents listed
in "agents", in which all
```

```
        * "validWorlds" are assumed to be true. Furthermore, it will return an
updated reference to the
        * "currentWorld" which will point at the newly created world. (Rather
than the old one.)
        * @param validWorlds - the worlds in which this upgrade was valid
        * @param agents - the agents aware of the upgrade
        * @param currentWorld - the current world
        * @return - an updated reference to the current world
        */
       World performRadicalUpgrade(ArrayList<World> validWorlds,
ArrayList<Agent> agents, World currentWorld) {
              ArrayList<World> newWorldList = new ArrayList<World>();
              ArrayList<Relation> relList = new ArrayList<Relation>();
              ArrayList<World> targetList = new ArrayList<World>(worldList);
              World newCurrentWorld = currentWorld;
              // we want copies of all worlds for a private upgrade.
              for (World validWorld : targetList) {
                     for (long i = 0; ; i++) {
                           // if we exceed this value, we can't continue.
Should not happen in normal operation
                           // without extreme computing power, or it will take
literally years to calculate.
                           if (i >= Long.MAX_VALUE - 1) {
                                  throw new RuntimeException("Error: too much
duplicates of World \"" +
                                                validWorld.getName() + "\". Under
normal conditions, this error can only be "
                                                + "reached by formula's which will take
close to forever (years) to validate");
                           }
                           World myWorld = new World(validWorld);
                           // long + string = string in the form of 1234string
                           myWorld.setName(i + myWorld.getName());
                           // get the next number if this one already exists.
                           if (worldList.contains(myWorld)) {
                                  continue;
                           }
                           worldList.add(myWorld);
                           newWorldList.add(myWorld);
                           // if the current world is equal to the world we
just duplicated, set the duplicate as current.
                           if (validWorld.equals(currentWorld)) {
                                  newCurrentWorld = myWorld;
                           }
                           for (Agent myAgent : agentList) {
                                  // add reflexive relations for all agents.
                                  relList.add(new Relation(myAgent, myWorld,
myWorld));
                                  // if agent is not in the list, he believes
the old world is true.
                                  if (!agents.contains(myAgent)) {
                                         relList.add(new Relation(myAgent,
myWorld, validWorld));
                                  }
                           }
```

```
                        break; // if we are here, we are done for with this
world. (no more numbers needed.)
                  }
            }

            for (Relation target : relationList) {
                  if (target.getFromWorld().equals(target.getToWorld())) {
                        continue; // skip reflexive relations, they were
added before.
                  }
                  /*
                   * for all agents:
                   * valid - valid / invalid - invalid should be added.
                   * valid -> invalid  should be reversed and then added.
                   * invalid -> valid should be added.
                   */
                  World to;
                  World from;
                  // reverse these.
                  if (validWorlds.contains(target.getFromWorld()) &&
!validWorlds.contains(target.getToWorld())) {
                        to =
newWorldList.get(targetList.indexOf(target.getFromWorld()));
                        from =
newWorldList.get(targetList.indexOf(target.getToWorld()));
                  }
                  else {
                        from =
newWorldList.get(targetList.indexOf(target.getFromWorld()));
                        to =
newWorldList.get(targetList.indexOf(target.getToWorld()));
                  }
                  relList.add(new Relation(target.getAgent(), from, to));


                  /*
                   * for unlisted agents:
                   * arrows back from all new to all old.
                   */
                  if (!agents.contains(target.getAgent())) {
                        from =
newWorldList.get(targetList.indexOf(target.getFromWorld()));
                        to = target.getToWorld();
                        relList.add(new Relation(target.getAgent(), from,
to));
                  }
            }
            relationList.addAll(relList);

            // clear all equiv classes
            equivalenceClassList = new ArrayList<EquivalenceClass>();
            // and refill them.

            for (Relation rel : relationList) {
                  addToEquivalenceClass(rel);
```

```
			}
			return newCurrentWorld;
		}

		/**
		 * This method makes an Upgrade according to DEL logic.
		 * This method *changes* this state, so it is recommended to use
		 * only on a clone.
		 * @param validWorlds  The list of "valid" worlds to which all arrows
need to go.
		 */
		void performUpgrade(ArrayList<World> validWorlds) {
			ArrayList<Relation> relList = new ArrayList<Relation>();
			for (Relation rel : relationList) {
				if (validWorlds.contains(rel.getFromWorld()) &&
!validWorlds.contains(rel.getToWorld())) {
					Relation temp = new Relation(rel.getAgent(),
rel.getToWorld(), rel.getFromWorld());
					// if relationList has this one, it means relList
will also have/get it, don't need duplicates.
					if (relationList.contains(temp)) {
						continue;
					}
					relList.add(temp);
				}
				else {
					relList.add(rel);
				}
			}

			relationList = relList;
			// clear all equiv classes
			equivalenceClassList = new ArrayList<EquivalenceClass>();
			// and refill them.
			for (Relation rel : relationList) {
				addToEquivalenceClass(rel);
			}
		}

		/**
		 * This method makes an Upgrade according to DEL logic. But only for 1
agent.
		 * This method *changes* this state, so it is recommended to use
		 * only on a clone.
		 * @param validWorlds  The list of "valid" worlds to which all arrows
need to go.
		 */
		void performConvervativeUpgrade(ArrayList<World> validWorlds, Agent
agent) {
			ArrayList<Relation> relList = new ArrayList<Relation>();
			for (Relation rel : relationList) {
				if (!rel.getAgent().equals(agent)) {
					continue;
				}
```

```
                        if (validWorlds.contains(rel.getFromWorld()) &&
!validWorlds.contains(rel.getToWorld())) {
                                Relation temp = new Relation(rel.getAgent(),
rel.getFromWorld(), rel.getToWorld());
                                // if relationList has this one, it means relList
will also have/get it, don't need duplicates.
                                if (relationList.contains(temp)) {
                                        continue;
                                }
                                relList.add(temp);
                        }
                        else {
                                relList.add(rel);
                        }
                }
                relationList = relList;
                // clear all equiv classes
                equivalenceClassList = new ArrayList<EquivalenceClass>();
                // and refill them.
                for (Relation rel : relationList) {
                        addToEquivalenceClass(rel);
                }
        }

        /**
         * Method used to create a clone of this state. Mainly intended to be
used before performing an
         * update or upgrade, so the original can be restored.
         */
        public State clone() {
                try {
                        State clone = (State) super.clone();
                        clone.agentList = new ArrayList<Agent>(agentList);
                        clone.worldList = new ArrayList<World>(worldList);
                        clone.relationList = new
ArrayList<Relation>(relationList);
                        clone.equivalenceClassList = new
ArrayList<EquivalenceClass>(equivalenceClassList);
                        return clone;
                }
                catch (CloneNotSupportedException cnse) {
                        // can't happen, but tell Java that.
                        Util.debug("A cloning exception has occured in
State.clone.");
                        return null;
                }
        }
}
```

## nl/rug/ai/believer/model/Util.java

```
package nl.rug.ai.believer.model;

import java.io.FileInputStream;
import java.io.IOException;
import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.Properties;

/**
 * A utility class providing support functions that should be available in the
whole program, but do not really have
 * anything to do with any class.
 * @author S1849700
 *
 */
public class Util {

	static boolean verbose = false;
	static ArrayList<String> reservedKeywords = new ArrayList<String>();

	// Util contains a few constants used to evaluate.
	// If you add one, you must also add it's exact name to the
allKeyWords[] array, and the config.ini
	static String turnstyle = "|=";
	static String or = " v ";
	static String and = "^";
	static String implies = "->";
	static String equals = "<=>";
	static String not = "~";
	static String belief = "B";
	static String knowledge = "K";
	// used for conditional beliefs
	static String conditional = "CONDITIONAL";
	// used for updates.
	static String update = "UPDATE";
	static String radical_upgrade = "RADICAL_UPGRADE";
	static String conservative_upgrade = "CONSERVATIVE_UPGRADE";
	static String agentStartTag = "{";
	static String agentEndTag = "}";
	static String agentSeparator = ",";

	// used for reading a config file. These Strings *must* be identical to
the field names above.
	private static String allKeyWords[] = {"turnstyle", "or", "and",
"implies", "equals", "not",
			"belief", "knowledge", "conditional", "update",
"radical_upgrade", "conservative_upgrade",
			"agentStartTag", "agentEndTag", "agentSeparator"};

	// static "constructor".
	static {
		try {
			for (String temp : allKeyWords) {
```

```
                        // use reflection to obtain the fields matching
allKeyWords
                        Field field = Util.class.getDeclaredField(temp);
                        // get the content of that field and add it to our
array. This way we only need to add
                        // new fields in 2 places.
                        reservedKeywords.add((String)field.get(null));
                  }
            }
            catch (Exception e) {
                  e.printStackTrace();
                  // this "should" not go wrong unless someone broke the
allKeyWords list.
                  throw new Error("Internal failure, severe. If persistent,
contact " +
                              "the developer of this application.");
            }
      }


      /**
       * This method prints debug text if verbose is set. It has no access
modifier (default), and is this
       * public to all classes inside the package, but not outside.
       * @param text The text to print.
       */
      static void debug(String text) {
            if (verbose) {
                  System.out.println(text);
            }
      }

      /**
       * This method is responsible for reading the configuration file, and
loading all values found there
       * into this class.
       */
      static void loadConfig() {
            try {
                  ArrayList<String> currentOperators = new
ArrayList<String>();
                  Properties myProps = new Properties();
                  FileInputStream MyInputStream = new
FileInputStream("config.ini");
                  myProps.load(MyInputStream);
                  for (String word : allKeyWords) {
                        String value = myProps.getProperty(word);
                        if (value == null ) {
                              throw new RuntimeException("No value for: " +
word);
                        }
                        if (value.charAt(0) != '"' ||
value.charAt(value.length()-1) != '"') {
                              throw new RuntimeException("Invalid value
for: " + word);
```

```
                        }
                        value = value.substring(1, value.length()-1);
                        if (value.trim().equals("")) {
                              throw new RuntimeException("Invalid value
for: " + word);
                        }
                        for (String temp : currentOperators) {
                              // check if the new string is a subset of any
existing, or vice versa.
                              if (temp.indexOf(value) != -1 ||
value.indexOf(temp) != -1) {
                                    throw new RuntimeException("Value for
\"" + word + "\" (" + value +
                                                ") is a subset of \"" +
temp + "\" or vice versa");
                              }
                        }
                        currentOperators.add(value);
                  }
                  // if we get here, all entries are there and valid.
                  try {
                        for (int i = 0; i < allKeyWords.length; i++) {
                              // use reflection to obtain the fields
matching allKeyWords
                              Field field =
Util.class.getDeclaredField(allKeyWords[i]);
                              // overwrite security, allow us (through
reflection only) to write non public fields.
                              field.setAccessible(true);
                              // write the field. First param is null
because we write to a static field.
                              field.set(null, currentOperators.get(i));
                        }
                  }
                  catch (Exception e) { // this "should" not go wrong unless
someone broke the allKeyWords list.
                        e.printStackTrace();
                        throw new Error("Internal failure, severe. If
persistent, contact " +
                                    "the developer of this application.");
                  }
                  reservedKeywords = currentOperators;
            }
            catch (IOException ioe) {
                  throw new RuntimeException("Could not read file:
config.ini");
            }
      }
}
```

## nl/rug/ai/believer/model/World.java

```
package nl.rug.ai.believer.model;

import java.util.ArrayList;

/**
 * This class represents a world. Each world has a name, and an ArrayList of
values that represent
 * propositional letters than hold true in this world.
 * @author S1849700
 *
 */
public class World {

	private String name;
	private ArrayList<String> values = new ArrayList<String>();

	/**
	 * constructor, used to set up the world
	 * @param name - desired name of the world
	 */
	World(String name) {
		if (name == null) {
			throw new NullPointerException("Contstructor name of World
was null");
		}
		this.name = name;
	}

	/**
	 * copy constructor
	 * @param other - the world to copy
	 */
	World(World other) {
		name = new String(other.name);
		values = new ArrayList<String>(other.values);
	}

	/**
	 * accessor method for name
	 * @return the name of the world
	 */
	public String getName() {
		return name;
	}

	/**
	 * Be sure this is only called on a world that is not added to any state
yet.
	 * @param name
	 */
	void setName(String name) {
		this.name = name;
	}
```

```
	/**
	 * method used to add a proposition to this world. All added
propositions are
	 * always true.
	 * @param value - the proposition to add.
	 */
	public void addProposition(String value) {
		if (value == null) {
			throw new NullPointerException();
		}
		for (String temp : Util.reservedKeywords) {
			// check if the new string contains any keyword.
			if (value.indexOf(temp) != -1) {
				throw new RuntimeException("Value for \"" + temp +
							"\" is a subset of \"" + value + "\"");
			}
		}
		values.add(value);
	}

	/**
	 * method used to remove (make false) a proposition again.
	 * @param value -- the proposition to be removed.
	 */
	public void removeProposition(String value) {
		values.remove(value);
	}

	/**
	 * Method used to check if a certain proposition is true or false in
this world.
	 * @param value - the proposition to check.
	 * @return -- true or false, depending on whether the proposition exists
or not. (true if it does)
	 */
	public boolean containsProposition(String value) {
		return values.contains(value);
	}

	/**
	 * We consider a World equal if their name is equal.
	 * @see Object#equals(Object)
	 */
	@Override
	public boolean equals(Object other) {
	    //check for self-comparison
	    if ( this == other ) {
	      return true;
	    }
	    // this renders an explicit check for "other == null" redundant,
since it does the check for
	    // null already - "null instanceof [type]" always returns false.
	    if (!(other instanceof World)) {
	      return false;
```

```
            }
            World otherWorld = (World)other;
            // we ignore values of this world here, as the name is supposed to
be a unique identifier.
            return otherWorld.name.equals(name);
        }

        /**
         * @see Object#hashCode(Object)
         */
        // must be overwritten if .equals is overwritten. (And should be based
on the same key fields.)
        @Override
        public int hashCode() {
              return name.hashCode();
        }
}
```

## nl/rug/ai/believer/model/exception/DuplicateException.java

```java
package nl.rug.ai.believer.model.exception;

/**
 * Exception class used to deal with duplicate entries in the model
 * @author S1849700
 *
 */
public class DuplicateException extends RuntimeException {

	/**
	 * standard exception constructor
	 * @see Exception
	 * @param message
	 */
	public DuplicateException(String message) {
		super(message);
	}

	private static final long serialVersionUID = 1;
}
```

## nl/rug/ai/believer/model/exception/InvalidAgentException.java

```java
package nl.rug.ai.believer.model.exception;

/**
 * Exception class used to deal with invalid agents
 * @author S1849700
 *
 */
public class InvalidAgentException extends RuntimeException {

	/**
	 * standard exception constructor
	 * @see Exception
	 * @param message
	 */
	public InvalidAgentException(String message) {
		super(message);
	}

	private static final long serialVersionUID = 1;
}
```

## nl/rug/ai/believer/model/exception/InvalidWorldException.java

```java
package nl.rug.ai.believer.model.exception;

/**
 * Exception class used to deal with invalid worlds
 * @author S1849700
 *
 */
public class InvalidWorldException extends RuntimeException {

	/**
	 * standard exception constructor
	 * @see Exception
	 * @param message
	 */
	public InvalidWorldException(String message) {
		super(message);
	}

	private static final long serialVersionUID = 1;

}
```

## nl/rug/ai/believer/ui/CommandLineInterface.java

```
package nl.rug.ai.believer.ui;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import nl.rug.ai.believer.model.Core;

/**
 * This is a simple command line interface for the Believer program
 * @author S1849700
 *
 */
public class CommandLineInterface {

	private boolean quit = false;

	/**
	 * Main method, this one will start up a command line reader, load the
config, set up the rest
	 * of the program, and parse new commands.
	 */
	public void run() {
		BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
		System.out.println("Welcome to Believer\n");
		loadConfig();
		while (!quit) {
			String nextLine;
			try {
				nextLine = reader.readLine();
			}
			catch (IOException ioe) {
				System.out.println("Fatal error while reading input,
terminating...");
				return;
			}
			parse(nextLine);
		}
	}

	/**
	 * Method used to parse new incoming commands.
	 * @param input - the new command
	 */
	private void parse(String input) {
		Core core = Core.getCore();
		String myInput = input.toLowerCase().trim();
		try {
			if (input.equals("help")) {
				printHelp();
			} else if (myInput.equals("list worlds")) {
				System.out.println(core.listWorlds());
```

```
                } else if (myInput.equals("list relations")) {
                        System.out.println(core.listRelations());
                } else if (myInput.equals("list agents")) {
                        System.out.println(core.listAgents());
                } else if (myInput.equals("list equivalence classes")) {
                        System.out.println(core.listEquivalenceClasses());
                } else if (myInput.startsWith("verbose")) {
                        if (myInput.substring(8).trim().equals("on")){
                                core.setVerbose(true);
                                System.out.println("Verbose mode on.");
                        } else {
                                core.setVerbose(false);
                                System.out.println("Verbose mode off.");
                        }
                } else if (myInput.startsWith("read file")) {
                        if (core.readFile(input.substring(10).trim())) {
                                System.out.println("File was read
successfully.");
                        }
                        else {
                                System.out.println("Failed to read file.");
                        }
                } else if (myInput.startsWith("evaluate")) {

                                if (core.evaluate(input.substring(9).trim()))
{
                                        System.out.println("This formula
evaluates as true");
                                } else {
                                        System.out.println("This formula
evaluates as false");
                                }

                } else if(myInput.equals("quit") ||
myInput.equals("exit")) {
                        quit = true;
                        System.out.println("Goodbye");
                } else {
                        System.out.println("Unknown command. See 'help' for
all available commands");
                }
            }
            catch (RuntimeException re) {
                  System.out.println("Error: " + re.getMessage());
            }
            System.out.println(""); // add an empty line.
      }

      /**
       * support method which prints all available commands.
       */
      private void printHelp(){
            System.out.println("Welcome to Believer\n");
            System.out.println("The following commands are available:");
            System.out.println("help -- this command");
```

```
			System.out.println("list worlds -- lists all worlds");
			System.out.println("list relations -- lists all relations");
			System.out.println("list agents -- lists all agents");
			System.out.println("list equivalence classes -- lists all
equivalence classes");
			System.out.println("verbose on/off -- sets verbose on or off");
			System.out.println("read file <file> -- reads a specified file");
			System.out.println("evaluate <formula> -- evaluates a specific
formula/statement");
			System.out.println("exit -- exits the application");
			System.out.println("quit -- alias of exit");
		}

		/**
		 * support method which is used to initially load the configuration
file.
		 */
		private void loadConfig() {
			try {
			Core.getCore().reloadConfig();
			}
			catch (RuntimeException rte) {
				System.out.println("Error loading config, using default
settings.");
				System.out.println("Error was: " + rte.getMessage());
			}
		}

		/**
		 * main point of entry for the program.
		 * @param args -- none
		 */
		public static void main(String[] args) {
			CommandLineInterface cli = new CommandLineInterface();
			cli.run();
		}
}
```

# Appendix C: Muddy Children input file

# in this file we define the initial situation for the muddy children puzzle.
# Worlds only define true information. In other words C1 means child one is clean, likewise, the lack of
# C1 (or NOT(C1) ) means child one is muddy.

# AGENTS tag is followed by a list of names separated by comma's, may span over multiple lines.
AGENTS
a1, a2, a3

# WORLDS are followed by a name and then a set of values inside a set of parenthesis (mandatory,
# but may be empty)
WORLDS
w1 (C1, C2, C3)
w2 (C1, C2)
w3 (C1, C3)
w4 (C1)
w5 (C2, C3)
# w6 is the real world
w6 (C2)
w7 (C3)
w8 ()

# RELATIONS is followed by an agent name, and then a table of X's and O's. There need to be as many horizontal
# rows as there are vertical. (Each world needs to be in both sets, otherwise reflexivity can't be met.
# You can leave out worlds not used by the agent. Each name or X/O needs to be separated by comma. X means a
# relation, O means no relation. Relations are read as from "world left of the X" to "world above it".
# additional spaces/tabs may be used for human readable formatting.
RELATIONS

a1

| | w1, | w2, | w3, | w4, | w5, | w6, | w7, | w8 |
|---|---|---|---|---|---|---|---|---|
| w1, | X, | O, | O, | O, | X, | O, | O, | O |
| w2, | O, | X, | O, | O, | O, | X, | O, | O |
| w3, | O, | O, | X, | O, | O, | O, | X, | O |
| w4, | O, | O, | O, | X, | O, | O, | O, | X |
| w5, | X, | O, | O, | O, | X, | O, | O, | O |
| w6, | O, | X, | O, | O, | O, | X, | O, | O |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| w7, | O, | O, | X, | O, | O, | O, | X, | O |
| w8, | O, | O, | O, | X, | O, | O, | O, | X |

a2

| | w1, | w2, | w3, | w4, | w5, | w6, | w7, | w8 |
|---|---|---|---|---|---|---|---|---|
| w1, | X, | O, | X, | O, | O, | O, | O, | O |
| w2, | O, | X, | O, | X, | O, | O, | O, | O |
| w3, | X, | O, | X, | O, | O, | O, | O, | O |
| w4, | O, | X, | O, | X, | O, | O, | O, | O |
| w5, | O, | O, | O, | O, | X, | O, | X, | O |
| w6, | O, | O, | O, | O, | O, | X, | O, | X |
| w7, | O, | O, | O, | O, | X, | O, | X, | O |
| w8, | O, | O, | O, | O, | O, | X, | O, | X |

a3

| | w1, | w2, | w3, | w4, | w5, | w6, | w7, | w8 |
|---|---|---|---|---|---|---|---|---|
| w1, | X, | X, | O, | O, | O, | O, | O, | O |
| w2, | X, | X, | O, | O, | O, | O, | O, | O |
| w3, | O, | O, | X, | X, | O, | O, | O, | O |
| w4, | O, | O, | X, | X, | O, | O, | O, | O |
| w5, | O, | O, | O, | O, | X, | X, | O, | O |
| w6, | O, | O, | O, | O, | X, | X, | O, | O |
| w7, | O, | O, | O, | O, | O, | O, | X, | X |
| w8, | O, | O, | O, | O, | O, | O, | X, | X |

# References

Alchourrón, C. E., Gärdenfors, P., and Makinson, D. (1985). On the logic of theory change: partial meet functions for contraction and revision. *Journal of Symbolic Logic 50*, 510–530.

Baltag, A. (2002). A Logic for Suspicious Players: Epistemic Actions and Belief–Updates in Games. *Bulletin of Economic Research,* Volume 54, Issue 1, pp. 1–45.

Baltag, A. & Smets, S. (2008). A Qualitative Theory of Dynamic Interactive Belief Revision. In Bonanno, G., van der Hoek, W., and Wooldridge, M., *eds.*, *Logic and the Foundations of Game and Decision Theory (LOFT7)*, volume 3 of Texts in Logic and Games, pages 13–60. Amsterdam University Press.

Baltag, A., Ditmarsch, H. P. van & Moss L. S. (2008). Epistemic Logic and Information Update. In P. Adriaans & J. van Benthem, *eds., Handbook of the Philosophy of Information*, Elsevier Science Publishers, Amsterdam.

Baltag, A., Moss, L.S. & Solecki, S. (1998). The Logic of Common Knowledge, Public Announcements, and Private Suspicions. In I. Gilboa (ed.), *Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge (TARK'98)*, pp. 43–56.

Bayes, T. (1764). An Essay Toward Solving a Problem in the Doctrine of Chances, *Philosophical Transactions of the Royal Society of London*, 53, pp. 370-418.

Benferhat, S., Dubois, D., Prade, H. & Williams, M.A. (2009). A general framework for revising belief bases using qualitative Jeffrey's rule, *Proceedings of the Ninth International Symposium on Logical Formalizations of Commonsense Reasoning*, pp 7-12

Boutilier, C., Friedman, N. & Halpern J. Y. (1998). Belief revision with unreliable observations. In *Proceedings, Fifteenth National Conference on Artificial Intelligence (AAAI '96)*, pp. 127–134.

Ditmarsch, H.P. van, Hoek, W. van der, & Kooi, B.P. (2007) *Dynamic Epistemic Logic,* Synthese Library 337. Springer, Berlin.

Eijk, J. van (2004). *Dynamic epistemic modeling*. CWI report SEN-E0424.

Friedman, N., & Halpern, J. Y. (1997). Modeling belief in dynamic systems. part I: foundations. *Artificial Intelligence*, *95*(2), 257–316.

Friedman, N. & Halpern, J. Y. (1999). Modeling belief in dynamic systems. Part II: revision and update. *Journal of A.I. Research 10*, 117–167.

Java SDK API. (2011). In *Java™ Platform, Standard Edition 6*

*API Specification*. Retrieved March 14, 2012, from http://docs.oracle.com/javase/6/docs/api/

Jeffrey, R. (1992). *Probability and the Art of Judgment*. New York: Cambridge University Press.

Goldszmidt, M. & Pearl, J. (1996). Qualitative probabilities for default reasoning, belief revision, and causal modeling. *Artificial Intelligence*, 84, 57–112.

Halpern, J. Y. & Tuttle, M. R. (1993). Knowledge, probability, and adversaries. *Journal of the ACM*, 40(4), 917–962.

Halpern, J.Y. & Vardi, M.Y. (1991) Model checking vs. theorem proving: a manifesto. *Proceedings KR- 91*, Boston, MA, 325 334.

Markov, A. A. (1971). Extension of the limit theorems of probability theory to a sum of variables connected in a chain. reprinted in Appendix B of: R. Howard. *Dynamic Probabilistic Systems, volume 1: Markov Chains*. John Wiley and Sons.

Moore, G.E. (1912). *Ethics*. Oxford University Press, 1912. Consulted edition: The Home University Library of Modern Knowledge, volume 54, OUP, 1947.

Pearl, J. (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann.

Zhao, J. & Osherson, D. (2010). Updating beliefs in light of uncertain evidence: Descriptive assessment of Jeffrey's rule, *Thinking & Reasoning*, 16: 4, pp. 288-307.

Spohn, W. (1987). Ordinal conditional functions: A dynamic theory of epistemic states. In W.L. Harper and B. Skyrms, eds., *Causation in Decision, Belief Change and Statistics*, vol. 2, pp. 105–134. D. Reidel.