Inconsistency Handling in Multi-Agent Systems

John Bruntse Larsen

Kongens Lyngby 2011
IMM-BSc-2011-9

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

At the time of writing, agent programming languages are still a new technology and general purpose agent systems are difficult to use. These systems are based on agents that reason and act according to a belief base. A typical bug in such agents are inconsistency in the belief base, which can make them act unexpectedly. In this project I work with revisioning of beliefs, to handle inconsistency automatically in Jason, a practical multi-agent system. I also experiment with paraconsistency in Jason and with possible applications of it.

Resumé

På dette tidspunkt er agent programmeringssprog stadig en ny teknologi, og generelle agent systemer er svære at anvende praktisk. Disse systemer er baseret på agenter, der tænker og handler ud fra en vidensbase. Et typisk problem i disse agenter er inkonsistens i deres vidensbaser, der kan få dem til at opføre sig uforudsigeligt. I dette projekt anvender jeg revidering af vidensbasen til at håndtere inkonsistens automatisk i Jason, der er et praktisk anvendeligt multi-agent system. Jeg eksperimenterer også med parakonsistens i Jason og eventuelle praktiske anvendelser.

Preface

This thesis is written as a 20 ECTS bachelor thesis for DTU Informatics and is required for obtaining a bachelor degree in computer science. The thesis started 31/1/2011 and ended 27/6/2011.
My background for this project is from the courses at DTU which are *02180 - Introduction to Artificial Intelligence*, *02285 - Artificial Intelligence and Multi-Agent Systems* and *02122 - Software Technology Project*. This gives me a background in predicate logic and practical use of it so I was able to understand most of the material I found.

My supervisor Jørgen Villadsen has been a great help with providing and discussing material about paraconsistency, contacting the authors of Jason with my questions and being a general support. The PhD students and the other professors at DTU Informatics have also been very inspiring about this topic though I will not present what I discussed with them in this thesis.

Lyngby, June 2011

John Bruntse Larsen

Contents

Chapter 1

Introduction

In agent oriented programming a program is made of agents which act according to intentions and a model of their world. In AI it can be a more intuitive way to make programs. The model is often based on logic and if the model is inconsistent, it can lead to problems.
Although inconsistency is well defined there is no definite way to handle it and often solving inconsistencies must be done by the programmer rather than automatically. In this section I present inconsistency with an informal example and show how it occurs in different kinds of knowledge based systems. I am focusing on predicate logic and at times I refer to it as simply *logic* unless other is specified.

1.1 Inconsistency

In traditional predicate logic inconsistency refers to the conjunction $A \wedge \neg A$ where A is an aribitary expression in predicate logic. The consequence of inconsistency depends on where it occurs though. In a knowledge base, the classic logical consequence $\models$ can be used to derive new literals that are entailed by the knowledge base, however if the knowledge base contains an inconsistence, then everything can be derived from it.

$$\{A, \neg A\} \models B \text{ where } A \text{ and } B \text{ are arbitary expressions}$$

In a logic where logical consequence has the above property, $\models$ is defined in [5] to be *explosive.*
Humans also have a tendency to become inconsistent, especially when they try to make up a lie as in this example. Suppose you were defence attorney and heard this story from a witness.

On the night of the murder, the power was off in the neighbourhood so I did not notice anything in the other apartment. I was simply watching television from my bed not knowing the horrible things that happened at the time.

This story should immediately make you scream "Hold it!" as there is a clear contradiction between the power being off and watching television (though the witness may defend this statement somehow). This seems simple to a human but it is not trivial how to make an automated system handle this.

1.2 Inconsistency in Practice

Inconsistency is not a problem for humans but in AI it can cause many problems. The following shows a few examples that are relevant to this project.
In inference engines *KB* is a knowledge base represented as a list of clauses. By using logical cosequence an inference engine can, provided a *KB* and a logical formula A tell if $KB \models A$.
An example would be an exploring robot in a dark cave where *KB* initially contains a (finite) set of axioms and rules that models the world. A rule for this robot could be that if it hears a human voice from a position, some human is at this position (who is possibly trapped in this cave and is calling for help). The rule can be expressed formally in predicate logic.

$$\forall p(\ \mathit{voiceAt(p)} \Rightarrow \mathit{humanAt(p)})$$

If the robot perceives *voiceAt(p)* for some position p, then it can derive *humanAt(p)* by logical consequence and it can be added to *KB*.
An inconsistency could occur if the robot also had the rule (maybe by a programming mistake).

$$\forall p(\ \mathit{silenceAt(p)} \Rightarrow \neg \mathit{humanAt(p)})$$

If the robot perceived *silenceAt(p)* because it did not hear a human here, it would then derive ¬*humanAt(p)* and add it to *KB*. If the human later decided to call out for help at p and the robot perceived this, then the robot would also add *humanAt(p)* to *KB* and it would now be inconsistent. The inference procedure based on logical consequence would no longer be useful as it could derive everything and the robot might start acting very strange.

1.2.1 Jason

Jason is a practical interpreter of the agent-oriented programming language AgentSpeak and it is the technology in focus of this project. It operates by using a belief base and goals for applying relevant plans made by the programmer. The belief base is essentially a list of logical predicates referred to as *beliefs*.
Jason assumes an open world meaning that if the belief base contains p it does not assume $\neg p$ as well. However Jason allows both negated and non-negated beliefs to occur in the belief base, so it easily becomes inconsistent by a programming error like in the previously described *KB* of the inference engine. When the belief base becomes inconsistent Jason will just use the oldest belief and as a result the behaviour becomes unpredictable.
It is also difficult to identify the problem without manually inspecting the belief base, which can be quite large.

1.2.2 PDDL and STRIPS

PDDL and STRIPS are both agent languages that are used for agents with automated planning. This is different from Jason where agents have pre-programmed plans. STRIPS is the original language and PDDL is an extension of it however there is no commonly used interpreter of the language. Details of the language can be found in [6].
They model the world state as a set of predicates, however generally they assume a closed world meaning that $\neg p$ is assumed if p does not occur in the model. In this way the model only contains positive predicates and querying $\neg p$ succeeds if the model does not contain p (like the NAF operator in Prolog).
Because they assume a closed world, they cannot become inconsistent in the classical way of having both p and $\neg p$. It might still contain contradictionary predicates such as $agentAt(P1)$ and $agentAt(P2)$ at the same time, which states that the agent is at multiple positions. I do not analyze this problem in PDDL in this project.

1.3 Project Content

The goal of this project is to focus on the Jason technology and the problems of logical consequence with an inconsistent agent. My goal of the project is

To implement a particular method for handling inconsistency based on revisioning the belief base and experiment with practical use of paraconsistency. Belief revision is the major focus and I research potential uses of both in future Jason programming.

In chapter 2 I analyse the problems in this project and present the background for my implementation. This chapter is quite long as much of my work has been about this analysis.
In chapter 3 I present the design of the implemented belief revision in Jason and in chapter 4 I present the work I did with a paraconsistent logic. In chapter 5 I show the test cases I used and the results I got. In chapter 6 I reflect upon these results and reflect upon the project in general.
Finally I conclude on my stated goal.

1.4 Terms

I use a few terms and ways of writing that would be nice to be familiar with.
AGM refers to Alchourron, Gärdenfors and Makinson who proposed the postulates of belief revision.
I also use a particular way of showing datatypes inspired by functional programming types.

$$dataType : memberType_1 * memberType_2 * ... * memberType_n$$

Where $dataType$ is the name of the datatype and $memberType_i$ is the type of member i. In this way the constructor of a datatype can be represented as a tuple

$$dataType(member_1, member_2, ..., member_n)$$

Where each $member_i$ has the type $memberType_i$.
A similar model is used for functions

$$functor : arg_1 * arg_2 * ... * arg_n \rightarrow resType$$

Where $functor$ is the name of the function, arg_i is the type of argument i and $resType$ is the type of the result. Function calls are shown like this

$$functor(arg_1, arg_2, ..., arg_n)$$

CHAPTER 2

Analysis

In this chapter I present the general principles of belief revision and paraconsistency as well as Jason which is the agent technology I will be working with. This means that I also explain Jason and how it is used before I go into details with how it works. I only deal with the parts relevant to this project.
Then I present the belief revision algorithm my implementation is based on and the paraconsistent logic I experiment with. Finally I summerize the problems I work with, found in this analysis.

2.1 Belief Revision

Informally this is about avoiding inconsistency completely by revisioning the belief base when an inconsistency occurs, so that this inconsistency is no longer present and can not be derived. One way to do this is to *contract* one of the beliefs that caused the inconsistency from the belief base. The contraction of a literal b from belief base B can be defined as such

$$B \models b \Rightarrow (B \dot{-} b) \not\models b$$

AGM proposed postulates for belief revision and contraction that should be satisfied by a contraction algorithm, however according to [3] they are generally

thought not to suit practical agents well.
The AGM style of belief revision is by *coherence* meaning that contraction of a belief b from a belief base B modifies B minimally so that B does not entail b. This means that beliefs derived from b are not necessarily removed.
Another kind of belief revision and contraction is by *reason-maintenance* where beliefs derived with the help of b are removed as well, unless they are justified in other ways. The idea is that beliefs with no justifying arguments should be removed, but it may remove beliefs that are not necessarily inconsistent with the belief base.

2.2 Paraconsistent Logic

A paraconsistent logic is a logic where logical consequence is not explosive. There are several such logics but there is no single logic that is found useful for all purposes and not all are designed to be useful in automated reasoning.
Paraconsistent logic in general is open to discussion and practical use of it is definitely interesting to research. More about paraconsistent logic can be found in [5].

2.3 Jason Language

I have already introduced Jason as a Java based interpreter of the agent oriented language AgentSpeak based on the BDI-model. I will not go into details of the BDI-model. In AgentSpeak plans are hard-coded in the agent program which makes planning in AgentSpeak very imperative and fast, but it also extends AgentSpeak with other features such as communication of plans and beliefs between agents.
A complete manual for Jason can be found in [1] but in this section I summarize the parts relevant to this project.

2.3.1 Belief

Beliefs are logical predicates which may or may not be negated (by using ~). In Jason a negated belief is said to be strongly negated which is different than weak negation, which in some systems are also called negation-as-failure. The difference can be illustrated like this.

Strong negation:	*It is not raining*	`~raining`
Weak negation:	*I do not believe it is raining*	`not raining`

In the first case I know for sure that it is not raining. In the second case I can only tell that I currently do not believe it is raining but I do not reject the possibility. Agents that use strong negation assume an open world, while those that do not, assume a closed world. In Jason all beliefs are kept in a belief base, where they can be interpreted to be in conjunction. Querying a belief succeeds if it can be unified with a belief in the belief base (or if it can not be unified in the case of weak negation).

2.3.2 Rule

Rules in Jason are very similar to Prolog clauses both in their form and how they are used. They can (sort of) be interpreted as definite clauses in first order logic. This is an example of a Jason rule.

```
c :- a & b
```

Where the head/positive literal is *c* and the body/negative literals is `a & b`. However it is also possible to make a rule with strong negation of any of the literals.

```
~c :- ~a & b
```

So that the analogy of rules as clauses is a bit more difficult (if `~a` and `b` is true then `~c` is true). Also the body can be any logical formula using a set of logical operators.
Rules are often used when checking plan contexts or with test goals to compute a particular unification like in Prolog. Because of this, rules are also a part of the belief base, however they can not be added or removed dynamically like beliefs.
Querying a rule succeeds if querying the body succeeds. Querying the body succeeds depending on the logical formula of the body.

2.3.3 Goals

Goals represents the intentions of the agents and the agent applies plans that matches the current goals. Jason distinguish between test and achievement goals but it is not relevant to this project to understand the difference. Goals are not part of the belief base and as such they can not make the agent inconsistent.

2.3.4 Plans

Plans are very important in this project as they describe which actions to use and how added beliefs depend on each other. Unlike automated planning languages like PDDL, AgentSpeak (and Jason) agents use a database of pre-compiled plans for the planning. It plans by reacting to events caused by adding/removing beliefs and goals.
A plan in Jason has an optional label, a trigger event, a context and a plan body. The parts in brackets are optional.

```
[@<label>] <trigger event> : <context> <- <body>
```

The label is a predicate which can be used to either just name the plan or annotate the plan for actual appliances.
The trigger event can be a belief or goal that is added or deleted.
The context is a logical formula that succeeds if and only if the plan is *applicable.* Jason uses the plan body of the first found applicable plan. The logical formula may contain beliefs, rules or even internal actions that can succeed or not.
The plan body is a series of actions that the agent will use to carry out the plan. Actions could be adding/removing goals or beliefs. A plan succeeds if each action succeed in the plan. Plans are often used in a recursive way such that the agent is reactive.
Although plans and rules look similar, they should be distinguished as they are used in very different ways. First and foremost rules are part of the belief base and plans are not. Rules can not cause actions either.

2.3.5 Annotations

Many language constructs in Jason can be annotated with a list of terms. This can be used for flagging beliefs, plans and goals with extra information. As default all beliefs are annotated with the source which tells where the agent got the belief from. Percieved beliefs are annotated with `source(percept)`, beliefs added by the agent itself (called *mental notes* in [1]) are annotated with `source(self)` and beliefs from other agents are annotated with `source(<agent>)` where `<agent>` is the name of the agent who sent it.

2.3.6 Communication

The final relevant feature is communication between agents. Agents can use an internal action to communicate in many ways but it is mostly the ability to tell other agents new beliefs through messages that is interesting in this project. While this can be useful it is also a potential source of inconsistency that needs to be handled.

2.4 Jason Architecture

Jason will be the development platform of this study, which is why it is important to understand the code behind it. I will only try to explain the classes that are relevant to this project and implementation. These are shown in figure 2.1 and I will explain some of these parts in detail. The explanations and the figure are based on both the Jason documentation and reading the source code.

2.4.1 Agent

This class represents the Jason agent that runs the code. Figure 2.1 shows that it is in a way central in the architecture, by using classes from every Java package. The Agent defines two methods of interest with regard to this project. The first method is **buf** the *Belief Update Function* which is only used for updating percepts. It takes as argument the list of current percepts which are added to the belief base and removes the old percepts not in this list. The percepts are received from the environment.
The second method is **brf**, the *Belief Revision Function* which is used for every other modification of the belief base. It takes as arguments the belief to add, the belief to remove and the intention behind this belief revision. In the default agent the belief base consistency is not checked at all, so the belief base can be inconsistent when this method has finished.

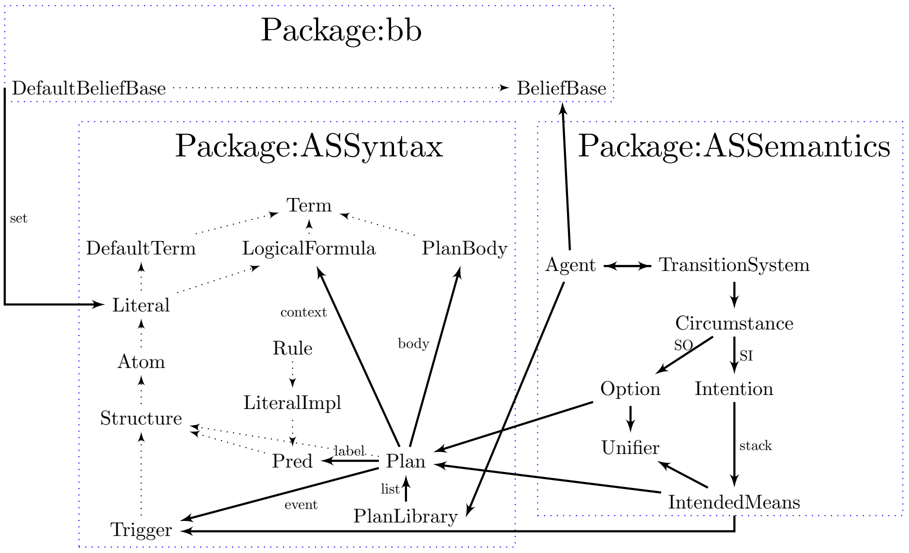


Figure 2.1: The relevant classes of Jason organized in Java packages. A filled arrow between two classes means that one class has the pointed class as a member. A dotted arrow between two classes means that one class extends or implement the class or interface pointed at.

It is very easy to make Jason use a customized version of this class, by extending it with a new agent in Java.

2.4.2 Term

The internal structure of beliefs and plans are defined in the ASSyntax package as seen in figure 2.1. The figure shows some interesting relations between the classes. It also shows how beliefs should be created internally an elegant and efficent way.
Beliefs and the body of plans are only related at the top level as a Term where it branches out with the interface LogicFormula for arbitary logical formulas (both beliefs and rules) and the interface PlanBody used for representing the body of a plan in Jason.

Beliefs and Rules

The DefaultTerm is the top abstract class of beliefs and rules and it branches out into special classes such as the NumberTerm for real numbers and ObjectTerm for Java classes, as well as the pure logical belief starting with the abstract Literal class. The branch of Literals are shown in table 2.1. The class ASSyntax defines methods for creating new literals that should be used rather than the constructors directly.
Although rules and beliefs are both instances of a Literal internally, they should be interpreted differently in Jason. When inspecting the belief base of an Agent you will not see which beliefs that the belief base entails according to the rules. This means that an agent may belive more than what the set of beliefs shows.

Class	Description	Datatype	Example
Literal	Abstract class of all literals	N/A	N/A
Atom	Positive literal in propositional logic	*String*	`p.`
Structure	Predicate in first order logic possibly with variables	*Term* list	`shape(earth,round).`
Pred	Adds annotations	*Term* list	`shape(earth,round) [source(self)].`
LiteralImpl	Adds strong negation	*boolean*	`~shape(earth,flat) [source(self)].`
Rule	Rules	*LogicalFormula*	`~hasEdge(X):- shape(X,round).`

Table 2.1: The table is ordered by the derived classes (Structure is derived from Atom etc.). It may seem weird that a rule is a literal but it means that the head is a literal.

Plans

Figure 2.1 shows that Plan extends Structure and so it is also a Literal. It consists of an optional label which is a Pred, the Trigger which also extends the Structure, the context which is a LogicalFormula and the body which is a PlanBody.

$$Plan : Pred * Trigger * LogicalFormula * PlanBody$$

The PlanBody represents both the current step in the plan and the tail of the plan to form a linked list. The current step has a type corresponding to the type of plan step (such as `!`, `?`, `+` or `-`) . All types are defined as an enum BodyType.

$$BodyType = \{none,\ action,\ internalAction,\ beliefAddition, ...\}$$

The PlanBody interface is implemented in the class PlanBodyImpl which extends Structure. The Term is the current step of the plan and the PlanBody is the tail of the plan.

$$PlanBodyImpl : Term * PlanBody * BodyType$$

2.4.3 TransitionSystem

The agent updates and uses a belief base to reason and plan for an intention. This behaviour is defined in the TransitionSystem. The relevant part is where it revises the belief base according to the current intention.
Figure 2.1 shows that each agent is assigned a TransitionSystem and each TransitionSystem is assigned a Circumstance which defines the currently selected Intention and Option. The Intention also tells what unifer was used to apply the plan.

2.4.4 Logical Consequence

In Jason logical consequence is defined by the method *logicalConsequence* in the interface LogicalFormula implemented by the Literal class. It takes as arguments the agent with a belief base and the initial unifier. The resulting sequence of unifiers is a potentially infinite sequence evaluated lazily.

$$logicalConsequence : Agent * Unifier \rightarrow Unifier \text{ sequence}$$

The method uses a backtracking algorithm to decide if $bb \vdash l$. The resulting unifiers θ can be characterized by a somewhat complex predicate logic expression I made, where $subs(l, \theta)$ is a function that substitutes the free variables of l with the corresponding substitution in θ.

$$bb \vdash l \Rightarrow$$

$$(\exists\theta(subs(l, \theta) \in bb))\vee$$

$$(\exists\theta, rule(rule \in bb \wedge subs(head(rule), \theta) = l \wedge bb \vdash subs(body(rule), \theta))$$

The point is that bb only proves l if a unifier can be found such that l occurs in bb either as a belief or as the head of a rule, where the body can be proved under this unifier. It can not prove something that does not occur in bb somehow, even if bb is inconsistent. In this way logical consequence in Jason is not explosive and thus it is paraconsistent.

2.5 Background of Belief Revision

The implementation is based on the work in [4] and [3] where the authors present a polynomial time algorithm for solving inconsistencies in AgentSpeak based on logical contraction as defined by AGM. They present an algorithm for contraction and suggestions for implementing belief revision in Jason. They state that it was not implemented.

2.5.1 Coherence and reason-maintenance

On page 69 in [3] they claim that the algorithm they use for contraction of beliefs can support both coherence and reason-maintenance without increasing the complexity.
Depending on the circumstances both styles can be useful. For example if b was a percept that was no longer perceived then a belief b' derived from b could still be true. The idea can be illustrated with this example.

Just because I cover my eyes the world could still exist.

However if I used b as the only argument for b' and b was later found to be incorrect, I can no longer claim b', as seen in this example.

I believed that I could reach the end of the world because it was flat. However when I found out the world was round, I could see that this could never happen and I dropped this belief.

2.5.2 Revision Algorithm

My belief revision is based on the algorithm shown in [4] which uses the term "apply rules to quiescence". This is related to the idea of closing under logical consequence and it means that you apply rules until no more beliefs can be added. The algorithm is shown in algorithm 1. I found that the principle of

Algorithm 1 revision by belief A in belief base K:

```
add $A$ to $K$
apply rules to quiescence
while $K$ contains a pair $(B, \neg B)$ do
    contract by the least preferred member of the pair
end while
```

closing under logical consequence does not translate well to Jason neither as rules or plans. This algorithm requires contraction of beliefs and in [4] they present an algorithm for this and show it has polynomial complexity.

2.5.3 Contraction Algorithm

In [4] they show that five of the AGM postulates of contraction are satisfied by their algorithm. It is not in the scope of my work to investigate these postulates further. The algorithm is shown in algorithm 2. The contraction uses a *justification* (l, s) which consists of a belief l and a support list of beliefs s, which is used by the contraction algorithm. Here l is the justified belief and s is the *support list*, the conjuncture of beliefs that was used to derive this belief. If one of the beliefs in the support list is false, the justification no longer justifies the belief. If a justification of a belief has an empty support list, then the belief is *independent*.
They define a directed graph where beliefs and justifications are nodes. Each belief has an outgoing edge the justifications where it occurs in the support list and an incoming edge from the justifications that justifies the belief. I have tried to illustrate it in figure 2.2.
Each support list s has a *least preferred member* $w(s)$ which is the belief that is the first to give up when contracting the belief that the justification justifies. They present a method to compute $w(s)$ however it is only a supplementing suggestion and $w(s)$ is supposed to be customizable by the programmer.

They show that this algorithm has complexity $O(rk+n)$ where r is the number of plans, k is the longest support list and n is the number of beliefs in the belief base. Reason-maintenance (removal of beliefs with no justifications) does not increase complexity of this either.

Algorithm 2 contract(l):

```
for all outgoing edges of $l$ to a justification $j$ do
  remove $j$
end for
for all incoming edges of $l$ from a justification $(l, s)$ do
  if s is empty then
    remove $(l, s)$
  else
    contract (w($s$))
  end if
end for
remove l
```

2.5.4 Declarative-Rule Plans

In [3] they remark that this method limits the format of plans to $te : l_1 \wedge ... \wedge l_n \leftarrow bd$, where te is a triggering event, $l_1 \wedge ... \wedge l_n$ is the context and bd is the plan body with a belief addition. Rather than limiting all plans in this way they instead define a *declarative-rule plan* that is a plan in this format especially used for belief revision.

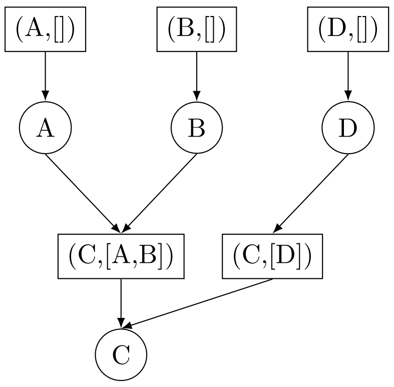
2.5.5 Implementing in Jason

In [3] they define the outgoing and incoming edges as two lists, such that *justifies* is the list of outgoing justifications and *dependencies* is the list of incoming justifications. If the plan of the intention is a declarative-rule plan $te : l_1 \wedge ... \wedge l_n \leftarrow bd$ with $+bl$ as the head of bd, the justification will be (bl, s) where the support list s will be

$$s = \begin{cases} [l_1, ..., l_n, \mathit{literal}(te)] & \text{if } te \text{ is a belief addition} \\ [l_1, ..., l_n] & \text{otherwise} \end{cases}$$

They suggest that lists of literals are stored by using annotations such as `dep([...]),just([...])`. I have tried to illustrate the graph of the belief base when using justifications in figure 2.2 where the beliefs and justifications are shown as nodes. It would also be possible to only keep the belief nodes but then there has to be multible copies of a belief; one for each justification in the dependencies list.

Finally they define the belief update function of the agent to update the justifications such that those with a deleted percept in them becomes independent.



(a) As graph

Belief	*justifies*	*dependencies*
A	[(C,[A,B])]	[(A,[])]
B	[(C,[A,B])]	[(B,[])]
D	[]	[(C,[A,B])]
C	[]	[(C,[A,B]),(D,[])]

(b) By using lists

Figure 2.2: The same belief base represented as a graph and by lists

2.5.6 Example and Limitations

The paper also presents a motivating example, where automated belief revisioning simplifies the process of solving an inconsistency for the programmer. In the example they note that while reason-maintenance is a nice property there, it is sometimes better to leave beliefs with no justifications in the belief base.
They also recognize that the solution has limitations. A particular interesting limitation, at least in my opinion, is the limited format of plans that can be used by the belief revision.

2.6 Multi-Valued Logic

The paraconsistent logic I consider is presented in [8] as a many-valued logic. Truth values are then not only *true* and *false* but can have as many values as necessary. This is similar to fuzzy logic where a truth value is a real number between 0 and 1 that denotes "how" true the truth value is, however this logic has concrete definitions of the logical operators and their truth tables.

2.6.1 Practical Example

In [8] use of the logic is demonstrated in a medical setting where the belief base is a combination of symptoms of two patients *John* and *Mary* and the combined knowledge of two doctors about two mutually exclusive diseases. However if classical predicate logical consequence is used to make a diagnosis, then *John* suffers from both diseases and because of this inconsistency, *Mary*, who both doctors agree on, also has both diseases. By using the multi-valued logic only *John* has this problem and *Mary* gets a consistent diagnose.

2.6.2 Logical Operators

In [8] several logical operators are defined but here I will focus on negation, conjunction, disjunction, biimplication and implication which are very commonly used in classical logic.

$$\neg p = \begin{cases} \mathit{false} & \text{if} \, p = \mathit{true} \\ \mathit{true} & \text{if} \, p = \mathit{false} \\ p & \text{otherwise} \end{cases}$$

$$p \wedge q = \begin{cases} p & \text{if} \, p = q \\ q & \text{if} \, p = \mathit{true} \\ p & \text{if} \, q = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$p \vee q \equiv \neg(\neg p \wedge \neg q)$$

$$p \leftrightarrow q = \begin{cases} \mathit{true} & \text{if} \, p = q \\ q & \text{if} \, p = \mathit{true} \\ p & \text{if} \, q = \mathit{true} \\ \neg q & \text{if} \, p = \mathit{false} \\ \neg p & \text{if} \, q = \mathit{false} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$p \rightarrow q \equiv p \leftrightarrow p \wedge q$$

One advantage of these definitions is that they are very simple to express in a functional or logic programming language. Jason has some logical programming through the use of rules and it would be interesting to see how it can handle this paraconsistent logic.

2.7 Problem Specifications

I have now shown the background of my implementation which focuses on the belief revision. My task is then to

- Extend Jason with an agent that can perform belief revision as described in the paper, however the agent should also implement it in a generalized way that a domain specific agent can override.
- The implementation should have some kind of debugging interface that shows how the belief revision occurs. This is important for practical use of belief revision.
- Address the restriction that belief revision can only be performed with declarative-rule plans as defined in the paper.
- Make the agent able to do belief revision with both coherence style and reason-maintenance.
- Give examples that shows uses of belief revision.

I will work with the multi-valued logic where I plan to explore potential uses in programming by

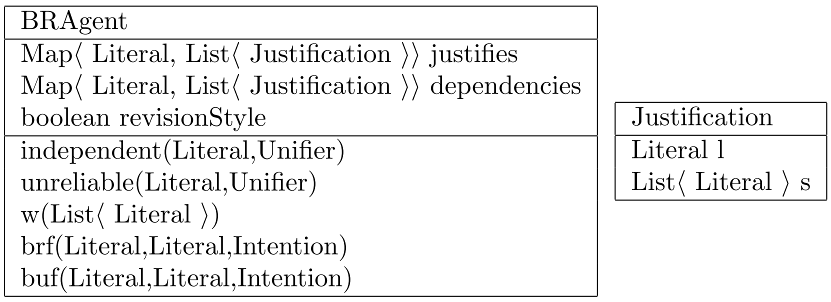
- Defining the semantics in Jason.
- Exploring how Jason can understand the multi-valued logic.

As shown earlier, logic consequence in Jason is already paraconsistent and I will experiment with possible uses of this.

Chapter 3

Design of the Belief Revision

The implemented design is based on what was proposed in [3]. I present my design of the belief revision that adresses the problems presented in the analysis. An overview of the implemented classes can be found in figure 3.1.



BRAgent
Map⟨ Literal, List⟨ Justification ⟩⟩ justifies
Map⟨ Literal, List⟨ Justification ⟩⟩ dependencies
boolean revisionStyle
independent(Literal,Unifier)
unreliable(Literal,Unifier)
w(List⟨ Literal ⟩)
brf(Literal,Literal,Intention)
buf(Literal,Literal,Intention)
Justification
Literal l
List⟨ Literal ⟩ s

Figure 3.1: Overview of BRAgent and Justification with the most relevant fields and methods.

3.1 Justifications

The justifications of belief are a core part of the contraction algorithm and using the internal Jason classes they can be defined as the class Justification which corresponds to (l, s) as seen in the analysis. One advantage of using internal classes rather than annotations of beliefs to represent this structure is that the relevant beliefs can be accessed faster than by querying the belief base. Instead the justifications are stored in the extended agent. Note that using the Literal class as a member means that every derived class (including rules and plans) potentially can have a justification.

3.2 BRAgent

The default Agent is extended with a class BRAgent that stores the justifications and defines all functions of the belief revision so that it is an extension of Jason that does not require altering the existing classes. This agent is also intended to be overrided with a more domain-specific agent but it does provide belief revision based on the one presented in [3].
Both BRAgent and Justification are put in the Jason library *jason.jar* in the package *jason.consistent* such that they are always available to extending agents but they could have been kept outside.

3.2.1 Associating Literals with Justifications

BRAgent maps every Literal to the lists *justifies* and *dependencies* of Justifications. By using the existing hashing function, the lists for a specific Literal can be found with low complexity and I avoid altering the existing Jason classes.
Because I use a mapping from Literal the result depends on the annotations of the Literal but when a justification is made, the beliefs are found in the belief base including all annotations. This is to avoid that a time annotation introduced later will cause problems with finding the correct justification of a Literal. For the same reason beliefs, to delete are first found in the belief base. A consequence of this is that annotations cannot be deleted from beliefs without removing the entire belief.

3.2.2 Coherence and Reason Maintenance

By default belief revision is done with reason maintenance but plans with a label annotated with `coh` performs belief revision with coherence.
Using coherence makes literals independent if they lose all dependencies during the belief revision. This is useful if an inconsistency of one belief does not require beliefs derived from it to be removed.
The style is stored in the *revisionStyle* boolean.

3.3 Auxillery Definitions

The belief revision based on contraction uses a few important auxillery definitions that are defined in the BRAgent class.

3.3.1 Independency

In the paper they present independent beliefs as beliefs with a single justification with an empty support list. Such could be percepts that does not depend on any other beliefs to be derived but there is no definition of which beliefs are independent. A particular agent using belief revision may want to make other kinds of beliefs independent.
To control this behaviour I define an independency-function that defines whether or not a Literal should get an independent justification. Again this is not introduced in the original paper but I will refer to Literals that fall within this definition as independent Literals and those that does not as dependent Literals.
I have implemented it as a function that tests if the Literal l is independent when added with Intention i. In the default BRAgent all Literals not added with a declarative-rule plan are independent.

$$independent(l, i) : Literal\ * Intention\ \rightarrow boolean$$

The way this function is used is shown in figure 3.2 although *setup* does not exist as an actual method in the code.

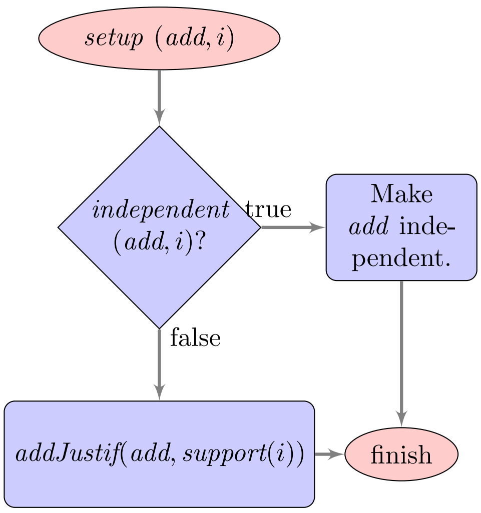


Figure 3.2: Adding a justification will update the justifies and dependencies lists of the literals in the justification using the internal mappings. Making a literal independent removes any previous justifications.

3.3.2 Reliability

The programmer of a domain specific agent might want to customize what should start the belief revision. To control this the BRAgent defines a reliability-function such that a belief revision occurs after adding an unreliable Literal. This is not introduced in the original paper but it adds further control of the belief revision.
I have implemented it as a function that tests if the Literal l is unreliable when added with intention i. In the default BRAgent all dependent Literals and all communicated Literals are unreliable.

$$unreliable(l, i) : Literal \ * \ Intention \ \rightarrow boolean$$

This function is not related to the worth of a literal presented in the paper or the trust-function which the agent uses to decide whether or not to ignore tell-messages from other agents. More about the trust-function can be found in [2].

```
@start[drp] +!start : a & b <- +c.
```

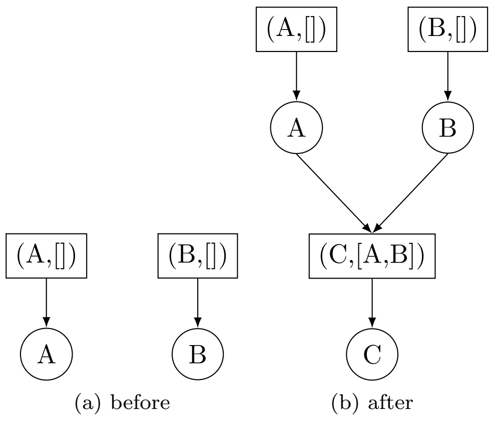


Figure 3.3: An example of a declarative-rule plan and the results.

3.3.3 Declarative-Rule Plans

As pointed out in the paper, declarative-rule plans used in belief revision must have a certain format such that the context is a conjunction of positive literals. In my implementation a declarative-rule plan that should use belief revision when adding or deleting a Literal must have a label annotated with `drp`. The added/deleted Literal is dependent and unreliable and an added Literal gets justifications according to the plan context and trigger event. The context is grounded with the unification applied by Jason. The result of using a declarative-rule plan is illustrated in figure 3.3.

Every belief is annotated with the time it was added. This annotation is updated if the belief is added later again.

3.3.4 Debugging

Plans with a label debug annotation `showBRF` prints the Literals in the belief base and their justifications before and after any belief revision in the plan. This printout represents the belief nodes and their justifications such as those in the graph of figure 2.2 in the analysis. It can be used to check the belief revision at run-time. An example of the debug output is shown in figure 3.4.

Any plan can be annotated with this label no matter if it is a declarative-rule

```
@start[drp,showBRF] +!start : a & b <- +c.
```

```
Belief base of agent before revision +c/{}
[agent] a[BBTime(1),source(self)], ([[]], [])
b[BBTime(2),source(self)], ([[]], [])

Belief base of agent after revision +c/{} using reason-maintenance
[agent] a[BBTime(1),source(self)], ([[]], [c])
b[BBTime(2),source(self)], ([[]], [c])
c[BBTime(3),source(self)], ([[a,b]], [])

-----
```

Figure 3.4: The result of applying this plan with debugging. It is assumed the agent is called `agent`. Before the revision `a` and `b` are independent beliefs and has a single justification with an empty support list while the list of justified literals is empty. After the revision, `c` occurs in both of their lists of justified literals while `c` has a single dependency with the support list [`a`, `b`]

plan or not.
It is also possible to annotate a belief with `showBRF` to show the results of that particular belief revision.
There is also an internal action that prints out this info but it can not show the belief base just before the revision with the new literal added.

3.3.5 Belief Preference $w(s)$

This function finds the least preferred Literal, in the list of literals s. It is used in the contraction algorithm to select a Literal to contract. The default definition selects the one with lowest rank.

1. Percepts have the highest rank
2. Mental notes have a higher rank than Literals with other sources except percepts
3. If the source of two Literals have the same rank, newer literals have a higher ranking than old literals.

The implementation of this ranking is trivial, although it uses the existing Jason functions and constants a lot. This function is designed to be overwritten by a domain specific agent.

3.4 Contraction

Figure 3.5 describes the contraction used in the solution and it is very similar to the one presented in [3]. Contracting a literal updates the justifications that refers to it. The details of the implementation are explained in the figure caption. I also define a method shown in figure 3.6 that contract Literals from the belief base so that it becomes consistent.

3.5 Belief Revision

The **brf** method derived from Agent is extended to implement the belief revision. This method is shown in figure 3.7. It sets up the added Literal with justifications as shown earlier and perform the belief revision.
Each belief revision has a cause Intention i which is usually either a specific plan or communication with another agent. It is also called with a a Literal to add or remove.
The method uses the independency-function and reliability-function to control the belief revision such that only these should be overwritten by a domain specific agent. To manage the size of figure it is split in more functions althogh not all of them exist as actual methods. The function that performs the revision is shown in figure 3.8.

3.6 Belief Update

Finally the belief update function **buf** is overwritten with a method that annotates all percepts with the added time and makes them independent before they are added to belief base in the usual manner. The method is shown in figure 3.9 and occurs as suggested in [3]. It is necessary to override this method as *brf* is not used at all for receiving percepts.

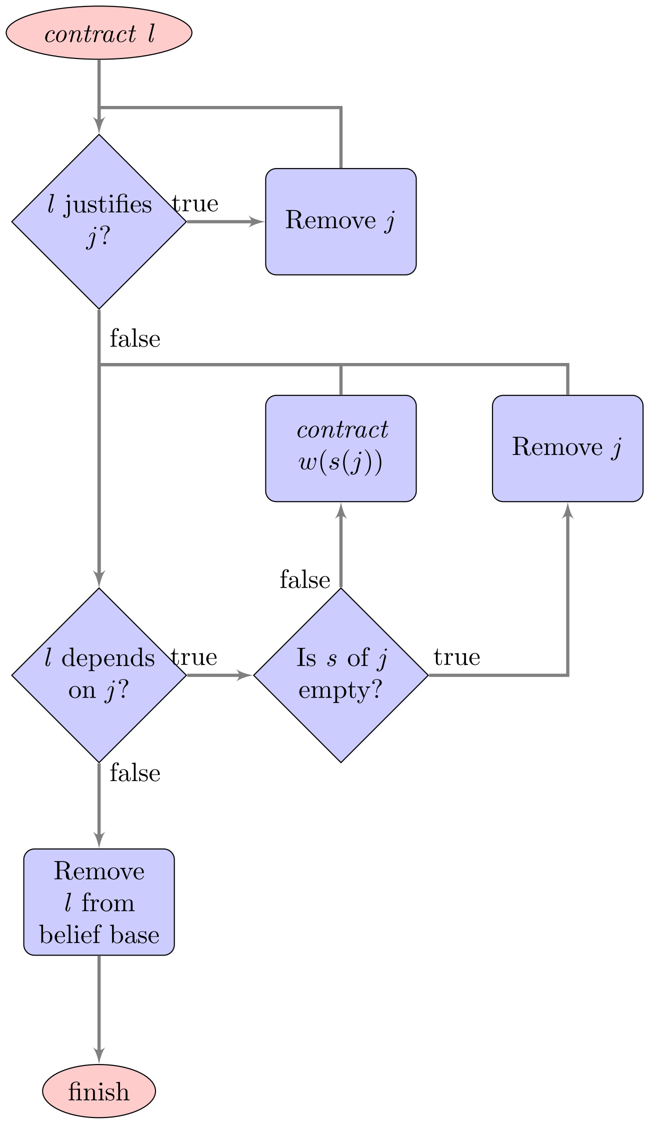


Figure 3.5: Flow chart of the contraction implementation. The function $s(j)$ denotes the support list of the justification j. Removing a justification updates the justifications of the referenced Literals. Literals with no dependencies left are either contracted as well or become independent Literals whether reason maintenance or coherence is used.

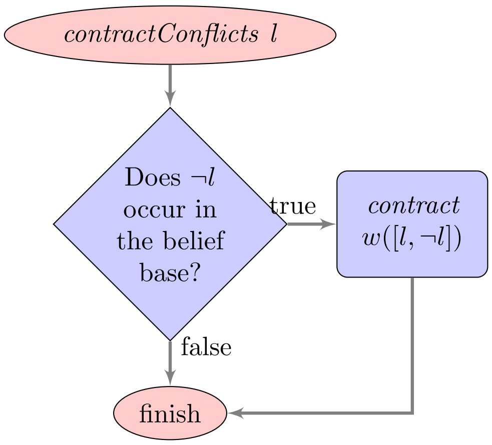

contractConflicts l
Does $\neg l$ occur in the belief base?
true
false
contract $w([l, \neg l])$
finish

Figure 3.6: Assuming the belief base was consistent before the revision it will be consistent after since at most one Literal l is added during the revision and either l or $\neg l$ will be removed after the revision.

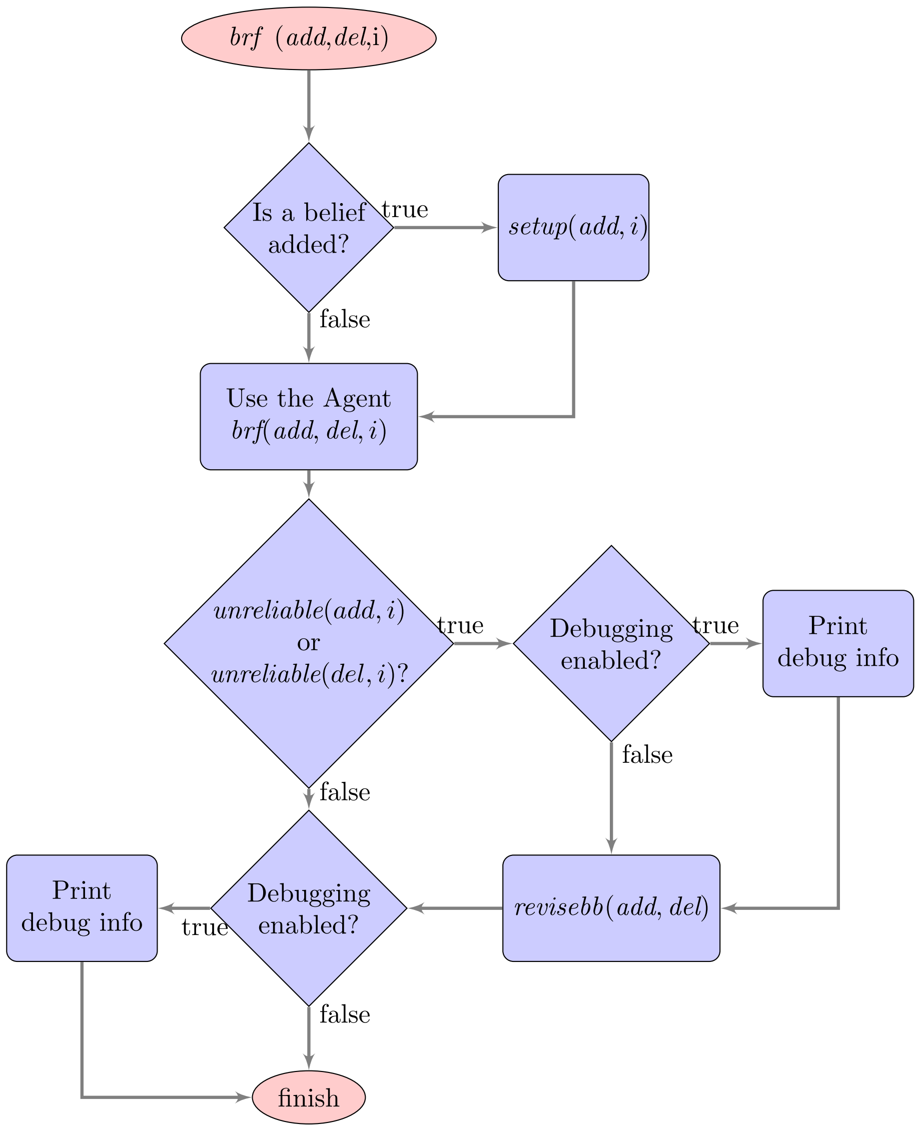


Figure 3.7: Belief revision where *add* is the literal to add, *del* is the literal to delete and *i* is the intention that caused this belief revision. Besides the belief revision itself there is also a control of the debugging printouts.

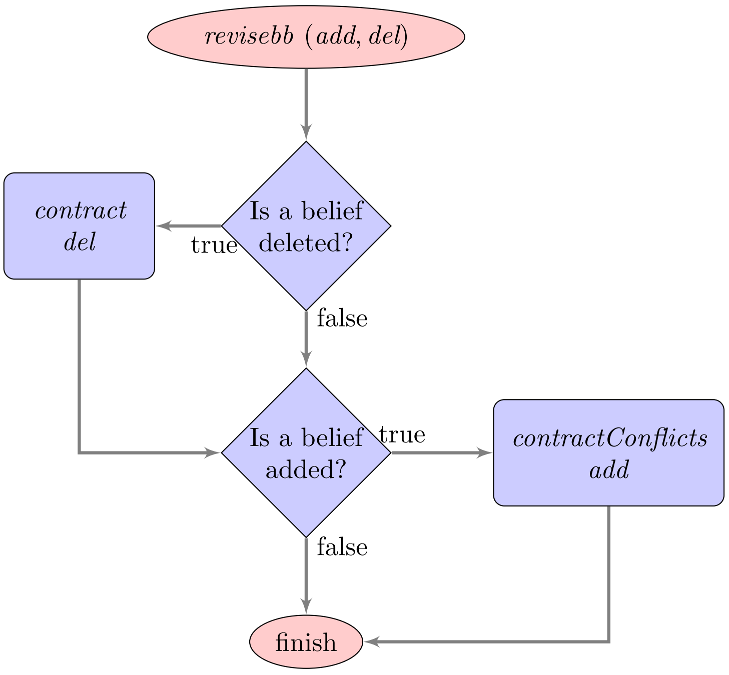


Figure 3.8: Revising the belief base and contracting any conflicts caused by removing or adding Literals. It does not actually exist as a java method.

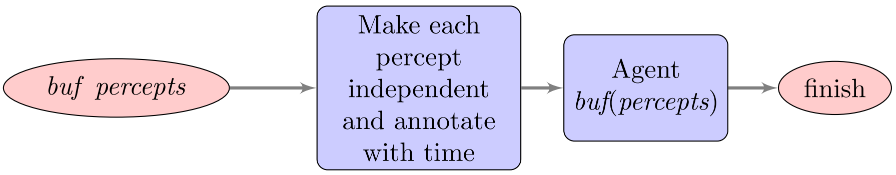


Figure 3.9: Each percept is made independent and is annotated with the current time before the usual belief update occurs.

Chapter 4

Design of the Paraconsistent Agent

I show how the multi-value logic presented in [8] can be implemented in Jason and explain how the existing paraconsistent logic conclusion analysed earlier can be used. The next section shows concrete examples.

4.1 Representing Multi-Value Logic

The implementation is based on logic programming such as in Prolog. Each definition in [8] can be expressed with one or several rules and beliefs and each agent using this logic must have these definitions in the belief base. Unlike Prolog there is no *cut* predicate so rules must exclude each other with more complex definitions. They are stil fairly short though. The relevant Jason language was shown in the analysis.

```
negate(t,f). negate(f,t).
negate(X,X) :- not X=t & not X=f.
opr(con,X,X,X).
opr(con,t,X,X) :- not X=t.
opr(con,X,t,X) :- not X=t.
opr(con,A,B,f) :- not A=B & not A=t & not B=t.
opr(eqv,X,X,t).
opr(eqv,t,X,X) :- not X=t.
opr(eqv,X,t,X) :- not X=t.
opr(eqv,f,X,R) :- not X=t & not X=f & negate(X,R).
opr(eqv,X,f,R) :- not X=t & not X=f & negate(X,R).
opr(eqv,A,B,f) :- not A=B & not A=t & not A=f & not B=t & not B=f.
opr(dis,A,B,R) :- negate(A,NA) & negate(B,NB) & opr(con,NA,NB,NR) & negate(NR,R).
opr(imp,A,B,R) :- opr(con,A,B,AB) & opr(eqv,A,AB,R).
```

4.2 Use of Multi-Value Logic

Any agent with these definitions is able to calculate a truth value using the multi-value logic. In a plan context or rule it can check whether truth values are as expected. The following examples shows how but they are not using the belief base and the plans would always succeed.

```
+p1 : negate(x,x) <- .print("~x is x").
+p2 : negate(f,X) & opr(con,X,x,x) <- .print("~f & x is x").
```

4.3 Inconsistent Belief Base

Recall that plans are applicable if and only if the context succeeds. By designing the plan contexts carefully it is possible to make the agent act with some rationality despite having an inconsistent belief base. I have not done a lot of work on such agents but there is a concrete example in the next section based on the case study in [8]. I translate each of the clauses in the knowledge base of the case study to beliefs and rules in Jason. This example shows how. The $\Box$ means that the truth-value is either *true* or *false* (no uncertainties about the symptoms).

$S_1x \wedge S_2x \rightarrow D_1x$ becomes `D1(X) :- S1(X) & S2(X).`

$\Box S_1 J$ becomes `S1(J).`

Chapter 5

Testing

In this section I will comment on the tests I made with both belief revision and paraconsistency. I explain the behaviour of the cases I found interesting, but the system (especially the belief revision) has been tested thoroughly.

5.1 Belief Revision

The test cases of belief revision has been divided into seven categories and in each category there are several cases. Every case except those in category 7 is implemented as a single agent and the beliefs of the tests have no real meaning.

5.1.1 Category 1, Propositional Logic

In these cases I only use beliefs in propositional logic and I test only with reason-maintenance style revision by adding beliefs. They are summerized in table 5.1 and all behave as expected.

Case	Purpose	Result
1a	$w(s)$ should return the oldest belief which will be removed.	The old belief is removed.
1b	Test of reason-maintenance with independent belief.	The independent belief and the belief justified by it are removed.
1c	Test reason-maintenance with dependent belief with no justified beliefs.	The dependent belief and one of the dependencies are removed.
1d	Test reason-maintenance with dependent belief with a justified belief.	The dependent belief, one of the dependencies and the justified beliefs are removed.

Table 5.1: Tests and results in category 1.

```
a(x).
b(x).
!start.
@start[drp,showBRF] +!start : a(X) & b(Y) <- +~a(Y).
```

Figure 5.1: Case 2a. The derived belief has both beliefs as dependencies. As result both `a(x)` and `~a(x)` are removed due to reason-maintenance.

5.1.2 Category 2, Predicate Logic

In these cases I have beliefs in predicate logic and I test only with reason-maintenance style revision due to adding beliefs. There are two cases.
In case 2a the belief base only contains grounded predicates such that the dependencies of a belief does not contain variables. The input and result of case 2a is shown in figure 5.1.
Case 2b is almost the same except that the context is replaced by a rule. One would expect that `~a(x)` is justified by the rule, which in turn is justified by the beliefs `a(x)` and `b(x)`. Reason-maintenance is not applied though as `~a(x)` remains after the revision. The input and result of case 2b is shown in figure 5.2.

```
a(x).
b(x).
~c(X,Y) :- a(X) & b(Y).
!start.
@start[drp,showBRF] +!start : ~c(X,Y) <- +~a(Y).
```

Figure 5.2: Case 2b. Reason-maintenance is not applied and the belief `~a(x)` gets null as a dependency.

```
a[source(self)].
a[source(other)].
!start.
@start[drp,showBRF] +!start <- -a[source(other)].
```

Figure 5.3: Case 3a. Belief `a` is removed entirely unlike in the default agent.

5.1.3 Category 3, Annotated Beliefs

Case 3a shows that removing an annotated belief removes the belief entirely. Input and result is shown in figure 5.3.
In case 3b an inconsistency occurs with an annotated belief which is not used for deriving anything, yet contracting it causes other beliefs to be removed. Case 3b is shown in figure 5.4.

5.1.4 Category 4, Coherence

In all previous tests I have used reason-maintenance as it is the default. By annotating beliefs with `coh` coherence style should be used instead. The cases are shown in table 5.2.
However in case 4b both the new and old belief in the inconsistency appears to have same time and other beliefs than expected are removed. The case is shown in figure 5.5.

```
a[annot1].
a[annot2].
b.
!start.
@start[drp] +!start : a[annot1] & b <- +c.
@c[drp,showBRF] +c <- +~a[annot2].
```

Figure 5.4: Case 3b. Inconsistency with annotated belief causes `a`, `c` and `~a` to be removed. One would expect only `a[annot2]`to be removed.

Case	Purpose	Result
4a	Coherence with inconsistent independent belief.	Only the contracted independent belief is removed.
4b	Coherence with inconsistent dependent belief.	Other beliefs than the expected are removed, see figure 5.5 .

Table 5.2: Cases of cateory 4. Case 4a goes as you would expect.

```
a.
b.
!start.
@start[drp] +!start : a & b <- +c.
@next[drp] +c <- +~c[coh,showBRF].
```

Figure 5.5: Case 4b. `c` and `~c` get the same time annotation and the revision only `b`remains.

Case	Purpose	Result
5a	With reason-maintenance. Beliefs with no justifications should be removed as well.	Reason-maintenance is applied as expected.
5b	With coherence. Beliefs with no justifications should become independent.	The belief dependent on the contracted belief becomes independent.

Table 5.3: Cases of cateory 5. All results are as expected.

Case	Purpose	Result
6a	Same belief added twice.	There is only one time annotation but it is updated.
6b	Same belief added twice but with different annotations.	There is only one time annotation but it is updated.

Table 5.4: Cases of cateory 6. All results are as expected.

5.1.5 Category 5, Removal of Beliefs

In the cases of category 3 there were a test with removal of annotated beliefs. In these cases I test that deleting beliefs updates their related justifications correctly. I test it in both reason-maintenance and coherence style. The cases are shown in table 5.3.

5.1.6 Category 6, Time Annotations

Here I test that the time annotation is updated when beliefs are added multiple times. I test it both when the exact same belief is added multiple times and when the belief is added a second time but with different annotations. The cases and results are show in 5.4

5.1.7 Category 7, External Belief Additions

All of the previous tests are carried out by making a single agent modify its own belief base by using plans however inconsistency is also very likely to occur in multi-agent systems where the agents communicate and it is worth testing belief revision in such an environment. Results are shown in table 5.5.
In case 7b an agent uses a communicated belief as a dependency of mental

Case	Purpose	Result
7a	Test $w(s)$ regarding communicated beliefs.	The mental note was kept over the communicated belief.
7b	Communicated belief as dependency of a higher rank mental note.	The old belief is removed.
7c	Dependencies across agents.	Dependencies does not carry between agents.
7d	Inconsistent by reliable source	Belief revision is not triggered and the agent remains inconsistent.

Table 5.5: Cases of cateory 7. Case 7a and 7d goes as you would expect.

note added with a declarative-rule plan. The mental note makes the agent inconsistent and while one might think the new mental note should be contracted because it depends on a source of low rank, the old mental note is contracted.
In case 7c the agent is made inconsistent by a belief told by another agent, however the reason it got that was because it told the agent about its own beliefs. The belief it told the other agent remains after the revision.
In case 7d is made inconsistent by a percept, but since percepts are a reliable source it is expected to remain inconsistent.

5.2 Multi-Value Logic

A truth table is shown by an agent with the definitions of the multi-valued logic, a goal and plan for each implemented operator and a helper test goal for calculating the truth values. The goals and plans for negation and conjunction are shown in figure 5.6. The other operators are tested in the same way.

```
!neg. !con.
+!neg <- ?negate(t,R1);?negate(f,R2);?negate(x,R3);
.print("neg: (t,",R1,"), (f,",R2,"), (x,",R3,")").
+?bi(O,R1,R2,R3,R4,R5,R6,R7,R8,R9) <-
?opr(O,t,t,R1);?opr(O,t,f,R2);?opr(O,t,x,R3);
?opr(O,f,t,R4);?opr(O,f,f,R5);?opr(O,f,x,R6);
?opr(O,x,t,R7);?opr(O,x,f,R8);?opr(O,x,x,R9).
+!con <- ?bi(con,R1,R2,R3,R4,R5,R6,R7,R8,R9);
?print(con,R1,R2,R3,R4,R5,R6,R7,R8,R9).
```

Figure 5.6: Multi-valued logic agent. Note that the print-plan simply print outs the given variables together with the corresponding truth values.

5.3 Doctor Example

This is the test case from [8] implemented as an agent in Jason.

```
s1(j). ~s2(j). s3(j). s4(j).
~s1(m). ~s2(m). s3(m). ~s4(m).
~d2(X):-d1(X). ~d1(X):-d2(X).
d1(X):-s1(X)&s2(X). d2(X):-s1(X)&s3(X).
d1(X):-s1(X)&s4(X). d2(X):-~s1(X)&s3(X).
!diagnoseJ. !diagnoseM.
+!diagnoseJ: d1(j) & d2(j) & ~d1(j) & ~d2(j) <-
  .print("j success").
+!diagnoseM: ~d1(m) & d2(m) & not d1(m) & not ~d2(m) <-
  .print("m success").
```

The plans show which beliefs that are/are not entailed by the belief base. It derives the same beliefs as with the multi-value logic in the paper.

$$bb \vdash d1(j), bb \vdash d2(j), bb \vdash \neg d1(j), bb \vdash \neg d2(j)$$

$$bb \nvdash d1(m), bb \vdash d2(m), bb \vdash \neg d1(m), bb \nvdash \neg d2(m)$$

Chapter 6

Discussion

The project has shown me a lot about practical use of both belief revision and paraconsistence. In this section I will discuss these things.

6.1 Belief Revision

Overall I have shown that the belief revision presented in [3] can be implemented in Jason without modifying the internal classes of Jason, however it required me to know the internal Jason architecture quite well to implement it in an efficient way, such that I used the existing code as much as I could. While the available documentation explained some parts it was often necessary to investigate the code in details to understand how to use the exisiting Jason architecture. I have presented the relevant parts in the analysis.
Putting the entire implementation in a single new class has the advantage of being compatible with older Jason agents and I tried to make the implementation customizable for domain specific agents. The default implementation gives the functionality they desired in [3].

6.1.1 Limitations

I have not added much functionality besides a few control mechanisms for coherence/reason-maintenance and debugging that was not present in [3]. This also means that the implementation has all limitations they acknowledged.
Programming an agent to use belief revision fully is difficult as it requires the agent to use declarative-rule plans. It remains a challenge to implement belief revision with an arbitary valid plan context.
The tests of category 7 with communicated beliefs and percepts shows that it is difficult for the agent to understand the dependencies of beliefs across agents. The plans that are used for communicating beliefs are implemented internally but according to the Jason manual [1], an agent can overwrite these plans. This could potentially be used to solve the problem but I have not investigated it much.
Annotations are generally problematic in the implementation. This is seen in the tests of category 3. If I instead did not use the belief in the belief base with all annotations, it would require the programmer to specify all annotations of every belief. This is not practical at all especially because the time annotation would have to be accurate to delete a belief. A solution might be to use a filtering function such that only some annotations are found in the belief base and the rest must be specified.
Test case 3b acts unexpectedly because the time annotation is not accurate enough. I currently use the internal system time in miliseconds and could easily use nanoseconds instead. It would then be less likely to occur again.
In Jason it is possible to use rules, arithmetic expressions, not-operations and internal actions in plan contexts which are not supported by the belief revision. Working with these could be an interesting and useful expansion.
Finally I did not get the time to set up a practical example showing the uses of belief revision as I wanted. I spend more time on cleaning up the implementation and generalize it for customization which I am also happy about. At least some of limitations I mentioned before could be solved by just extending the BRAgent with a new agent such that the original functionality is kept.

6.2 Paraconsistency

The tests showed that paraconsistent Jason agents have some practical uses without defining a new agent because logical conclusion is not explosive. This can be combined with the belief revision such that the agent can be inconsistent regarding some beliefs but still be consistent regarding others. It seems quite difficult to design an agent using this effectively though.

The multi-value logic can be expressed quite easily in Jason and could be the foundation for a knowledge base that defines logical consequence with this logic. As it is now it is only capable of evaluating truth values.

6.2.1 Limitations

Like seen in the doctor example a human is required to inspect that the agent has a problem of inconsistency towards one of the patients, and it is not able to solve the inconsistency by itself. The belief revision may be able to handle this to some extend by contracting beliefs but in this case it seems more likely that one of the rules should be removed rather than the beliefs, as the beliefs are more like percepts.
Although truth values of the multi-valued logic can be computed, the agent is unable to reason with these values. Doing this would require a new knowledge base that defined logical consequence with the multi-valued logic. In [7] it is shown how to make such a knowledge base in Prolog which might be possible to do in Jason as well using rules. Such an extension would be an interesting exercise.

Chapter 7

Conclusion

I have shown that automatic belief revision can be implemented in the multi-agent system Jason and that it can solve quite a few inconsistency problems. It does not act quite as expected but the design allows for some customized behaviour that future agents could use to improve the belief revision.
I have also shown how inconsistency can be handled by using paraconsistent agents in Jason and how Jason is able to interpret a paraconsistent multi-valued logic. This is illustrated with examples. The agent does not use the beliefs for reasoning with the mult logic. To do this one could make a belief base on top of Jason that defines logical consequence with the paraconsistent logic.

Appendix A

Code of Justification

```
package jason.consistent;

import jason.asSyntax.Literal;

import java.util.List;

public class Justification
{
   public Literal l;
   public List<Literal> s;

   public Justification(Literal l, List<Literal> s){ this.l
        = l; this.s = s; }

   public String toString(){
      return "("+l+","+s.toString()+")";
   }
}
```

Appendix B

Code of BRAgent

```java
package jason.consistent;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

import jason.JasonException;
import jason.RevisionFailedException;
import jason.asSemantics.Agent;
import jason.asSemantics.Intention;
import jason.asSyntax.ASSyntax;
import jason.asSyntax.ListTerm;
import jason.asSyntax.Literal;
import jason.asSyntax.LiteralImpl;
import jason.asSyntax.LogExpr;
import jason.asSyntax.LogicalFormula;
import jason.asSyntax.NumberTerm;
import jason.asSyntax.Plan;
import jason.asSyntax.RelExpr;
import jason.asSyntax.Structure;
import jason.asSyntax.Term;
import jason.asSyntax.Trigger.TEOperator;
import jason.asSyntax.Trigger.TEType;
import jason.bb.BeliefBase;

public class BRAgent extends Agent
{
   public static final boolean REASON_MAINT=true;
   public static final boolean COHERENCE=false;
   private static final Term drp = ASSyntax.createAtom("drp");
   private static final Term coh = ASSyntax.createAtom("coh");
   private static final Term debug = ASSyntax.createAtom("showBRF");

   private boolean revisionStyle=REASON_MAINT;
   private final long startTime = System.currentTimeMillis();

   // The Literal class has an effective hash function
   /** The list of literals in the support list*/
   private Map<Literal, List<Justification>> dependencies = new HashMap<Literal, List<Justification>>();
   /** The literals where this literal occur in their support list*/
   private Map<Literal, List<Justification>> justifies = new HashMap<Literal, List<Justification>>();

   /** the agent preference function. Returns the least preferred literal. */
   public Literal w(List<Literal> s) throws RevisionFailedException
   {
      Literal res = s.get(0);

      for (Literal l : s)
            if (isLowerQ(l,res)) res = l;

      return res;
   }

   /**
    * Standard method for determining literal quality based on source and age. Communicated literals
    * are lower quality than self derived literals. Self derived literals are lower quality than perceived literals.
    * Older literals are lower quality than new literals.
```

```
     * @throws JasonException If a literal is not annotated ↩
        with BBTime(n)
     */
    private boolean isLowerQ(Literal p, Literal q)
    {
       ListTerm qTimeList = q.getAnnots("BBTime");
       ListTerm pTimeList = p.getAnnots("BBTime");

       double qAge = ((NumberTerm)((Structure) qTimeList.get↩
           (0)).getTerm(0)).solve();
       double pAge = ((NumberTerm)((Structure) pTimeList.get↩
           (0)).getTerm(0)).solve();

       if (isLowerRank(p, q)) return true;
       else if (isSameRank(p, q) && pAge < qAge) return true↩
           ;

       return false;
    }

    private boolean isLowerRank(Literal p, Literal q)
    {
       // lower rank if q has a percept source and p does ↩
           not
       if (q.hasSource(BeliefBase.APercept)){
          if (p.hasSource(BeliefBase.APercept)) return false↩
              ;
       }
       // lower rank if q has a self source and p does not ↩
           have either self or percept source
       else if (q.hasSource(BeliefBase.ASelf)){
          if (p.hasSource(BeliefBase.APercept) || p.↩
              hasSource(BeliefBase.ASelf)) return false;
       }
       // p can not be lower rank if q is not a percept or ↩
           self obtained
       else return false;

       return true;
    }

    private boolean isSameRank(Literal p, Literal q)
    {
       // same rank if both contain a source of same rank
       if (q.hasSource(BeliefBase.APercept))
          return p.hasSource(BeliefBase.APercept);
       else if (p.hasSource(BeliefBase.APercept))
```

```
         return q.hasSource(BeliefBase.APercept);
      else if (q.hasSource(BeliefBase.ASelf))
         return p.hasSource(BeliefBase.ASelf);
      else if (p.hasSource(BeliefBase.ASelf))
         return q.hasSource(BeliefBase.ASelf);

      // all other sources have the same rank
      return true;
   }

   /** contract the belief base with respect to literal l ↩
       */
   private List<Literal> contract(Literal l) throws ↩
       RevisionFailedException
   {
      // l is removed
      List<Literal> res = new LinkedList<Literal>();
      res.add(l);

      // remove the justifications this literal justifies, ↩
          this list will be modified in each loop
      while (!justifies(l).isEmpty()){
         Justification x = justifies(l).get(0);
         remove(x);

         // if using reason maintenance, contract literals ↩
             with no other justifications
         if (revisionStyle==REASON_MAINT && dependencies(x.↩
             l).isEmpty())
            res.addAll(contract(x.l));
         //otherwise make the literal independent
         else if (revisionStyle==COHERENCE && dependencies(↩
             x.l).isEmpty())
            setupIndependentJust(x.l);
      }

      // make the literal underivable from the belief base,↩
           this list will be modified in each loop
      while (!dependencies(l).isEmpty()){
         Justification x = dependencies(l).get(0);
         if (!x.s.isEmpty())
            res.addAll(contract(w(x.s))); // contracting w(↩
                x.s) will remove x from l.dependencies
         else remove(x);                   // justifications↩
              with no support lists should just be removed
      }

      bb.remove(l);

      return res;
   }

   /** Remove a justification correctly (it loses all ↩
       references) */
   private void remove(Justification j) throws ↩
       RevisionFailedException
   {
      // remove the justification from the dependencies of ↩
          the justified literal
      dependencies(j.l).remove(j);

      // remove the justification from the justifies list ↩
          of every literal in the support list
      for (Literal x : j.s)
         justifies(x).remove(j);
   }

   /** Add a justification according to support list and ↩
       literal */
   private void add(Justification j){
      dependencies(j.l).add(j);
      for (Literal x : j.s)
         justifies(x).add(j);
   }

   /**
    * Find the instance of a literal in the belief base if ↩
        it exists.
    * Returns null otherwise.
    */
   private Literal getBelief(Literal l){ return bb.contains↩
       (l);   }

   /** resolve eventual conflicting literals by contraction↩
       */
   private List<Literal> contractConflicts(List<Literal> ↩
       res, Literal add) throws RevisionFailedException
   {
      // negate add as a new belief
      // using this constructor will not cache the hash ↩
          value
      Literal negated = //new LiteralImpl(add.negated(),add↩
          .getFunctor());
        ASSyntax.createLiteral(add.negated(), add.getFunctor(),add.getTermsArray());
        // add all terms, this will also reset the cached hash value
//      negated.addTerms(add.getTerms());

        // neg will be the belief in bb (with annotations and instantiated variables)
        Literal neg = getBelief(negated);

        // if a conflict was found, solve by contraction of least preferred literal
        if (neg != null){
            List<Literal> conflict = new LinkedList<Literal>();
            conflict.add(add);
            conflict.add(neg);
            res.addAll(contract(w(conflict)));
        }

        return res;
    }

    private List<Literal> groundLiterals(LogicalFormula fml) throws RevisionFailedException
    {
        List<Literal> res = new LinkedList<Literal>();

        if (fml instanceof RelExpr)
        {
            //NOTE: consider possibilities, if declarative rule-plans exists this plan was incorrect and the system can fail
            throw new RevisionFailedException("The context contained a relative expression");
        }
        else if (fml instanceof LogExpr){
            LogExpr lFml = (LogExpr) fml;
            if (lFml.getOp() != LogExpr.LogicalOp.and)
                throw new RevisionFailedException("Plan was not a drp");

            // assumes the logical operator is "&"
            LogicalFormula g = ((LogExpr) fml).getLHS();
            LogicalFormula h = ((LogExpr) fml).getRHS();

            res.addAll(groundLiterals(g));
            res.addAll(groundLiterals(h));
        }
        else if (fml instanceof LiteralImpl)
        {
            // apply the current unifier and add the ↩
                corresponding literal
            // with all annotations of the literal from BB
            fml.apply(ts.getC().getSelectedIntention().peek().↩
                getUnif());
            res.add(getBelief((LiteralImpl) fml));
        }

        return res;
    }

    /** Returns the support list of the justification of l ↩
        according to i */
    public List<Literal> supportList(Literal l, Intention i)↩
        throws RevisionFailedException{

        List<Literal> supportList;
        Plan p = i.peek().getPlan();

        supportList = groundLiterals(p.getContext());

        // add the trigger event in the case of a belief ↩
            addition
        if (p.getTrigger().getOperator() == TEOperator.add &&↩
             p.getTrigger().getType() == TEType.belief)
            supportList.add(getBelief(p.getTrigger().↩
                getLiteral()));

        return supportList;
    }

    /** Annotate a literal l with the current time. If it is↩
         already annotated
     * it will be updated.*/
    private void annotateTime(Literal l){

        Literal g = getBelief(l);
        if (g!=null){
            List<Term> time = g.getAnnots("BBTime");
            g.delAnnots(time);
        }

        l.addAnnot( ASSyntax.createStructure("BBTime",
```

```java
            ASSyntax.createNumber(System.currentTimeMillis() - startTime)));
   }

   /** Make the literal independent by giving it a single justification */
   private void setupIndependentJust(Literal l){
      dependencies(l).clear();
      add(new Justification(l,new LinkedList<Literal>()));
   }

   /** Return true if l is unreliable when added with intention I. <br>
    * As standard, dependent literals or if they are neither a percept
    * or self inferred are unreliable.*/
   public boolean unreliable(Literal l,Intention i){
      return !independent(l, i) ||
         l!=null && !(l.hasSource(BeliefBase.APercept) || l.hasSource(BeliefBase.ASelf));
   }

   /** Return true if l should be independent when added with intention i <br>
    * As default, all literals not added with a drp are independent */
   public boolean independent(Literal l, Intention i){
      Plan p = i!=null ? i.peek().getPlan() : null;
      return p==null || !p.getLabel().hasAnnot(drp);
   }

   /** returns true if any of the literals or the plan are annotated with "showBRF" */
   private boolean showBRF(Literal add, Literal del, Intention i){
      Plan p = i!=null ? i.peek().getPlan() : null;
      return p != null && p.getLabel().hasAnnot(debug) ||
            add!=null && add.hasAnnot(debug) ||
            del!=null && del.hasAnnot(debug);
   }

   /** returns true if any of the literals are annotated with "coh" */
   private boolean coherence(Literal add, Literal del){
      return add!=null && add.hasAnnot(coh) ||
            del!=null && del.hasAnnot(coh);
   }
```

```

    /** simply annotate percepts with time and make them ↩
        independent */
    @Override
    public void buf(List<Literal> percepts){
       if (percepts!=null)
          for (Literal l : percepts){
             annotateTime(l);
             setupIndependentJust(l);
          }
       super.buf(percepts);
    }

    /** Belief revision based on the article in the report. ↩
        Uses the definition
     * of declarative rule plans, which are the only plans ↩
         where the algorithm is
     * used. Such should have the annotation "drp" in the ↩
         label.
     * All other plans use the simple belief addition/↩
         deletion.<br>
     * Plans with labels annotated with "showBRF" or ↩
         literals annotated with "showBRF"
     * will have the results of belief revision printed<br>.
     * Literals annotated with "coh" will use coherence/AGM ↩
         style of belief revision*/
    @Override
    public List<Literal>[] brf(Literal beliefToAdd, Literal ↩
        beliefToDel, Intention i) throws ↩
        RevisionFailedException
    {
       // this must be done before revising bb (it can ↩
           change the literal)
       boolean independent = independent(beliefToAdd, i);
       boolean show = showBRF(beliefToAdd, beliefToDel, i);
       boolean unrely = unreliable(beliefToAdd,i) || ↩
           unreliable(beliefToDel,i);
       //set revision style COHERENCE=false, REASON_MAINT=↩
           true
       revisionStyle=!coherence(beliefToAdd, beliefToDel);

       // Every added belief will be annotated with time
       if (beliefToAdd != null){
          annotateTime(beliefToAdd);
          // setup justifications
          if (independent)
             setupIndependentJust(beliefToAdd);
         else
            add(new Justification(beliefToAdd, supportList(beliefToAdd,i)));
      }
      // disregard annotations when deleting beliefs (unlike the original jason agent)
      // by removing the entry including all annotations
      if (beliefToDel != null)
         beliefToDel = getBelief(beliefToDel);

      List<Literal>[] res = super.brf(beliefToAdd, beliefToDel, i);
      if (res == null)
         return res;

      // the list of beliefs deleted
      res[1] = new LinkedList<Literal>(res[1]);

      // only perform belief revision on unreliable beliefs
      if (unrely){
         if (show){
            System.out.println("Belief␣base␣of␣"+ts.getUserAgArch().getAgName()+
                  "␣before␣"+intentToString(i));
            getLogger().info(justifsToString());
         }

         // Contract the belief when it is removed, it will no longer be entailed
         if (beliefToDel != null)
            res[1].addAll(contract(beliefToDel));
         if (beliefToAdd != null)
            res[1] = contractConflicts(res[1], beliefToAdd);
      }

      if (show){
         System.out.println("Belief␣base␣of␣"+ts.getUserAgArch().getAgName()+
               "␣after␣"+intentToString(i)+
               "␣using␣"+(revisionStyle?"reason␣maintenance":"coherence"));
         getLogger().info(justifsToString());
         System.out.println("------");
      }

      // reset the revision style
      revisionStyle=REASON_MAINT;

      return res;
   }

   public String intentToString(Intention i){
      return (i!=null?"revision␣"+i.peek().getCurrentStep()↩
          :"init");
   }

   public String justifsToString(){
      String res = "";
      for (Literal l : bb)
         res+=l+",␣(["+depsToString(l)+"],␣["+justsToString↩
             (l)+"])\n";
      return res;
   }

   public String depsToString(Literal l){
      List<Justification> depends = dependencies.get(l);
      String deps = "";
      if (depends!=null && !depends.isEmpty()){
         deps=depends.get(0).s.toString();
         for (int i=1;i<depends.size();i++)
            deps+=","+depends.get(i).s;
      }
      return deps;
   }

   public String justsToString(Literal l){
      List<Justification> justifs = justifies.get(l);
      String justs = "";
      if (justifs!=null && !justifs.isEmpty()){
         justs=justifs.get(0).l.toString();
         for (int i=1;i<justifs.size();i++)
            justs+=","+justifs.get(i).l;
      }
      return justs;
   }

   /** Return the reference to the dependencies list of ↩
       this literal.
    * Modifying the returned list will also update the ↩
       entry. */
   public List<Justification> dependencies(Literal l){
      List<Justification> res = dependencies.get(l);

      // If the entry does not exist, make a new one
      if (res==null){
         res = new LinkedList<Justification>();
         dependencies.put(l,res);
      }

      return res;
   }

   /** Return the reference to the justifies list of this ↩
       literal.
    *  Modifying the returned list will also update the ↩
        entry. */
   public List<Justification> justifies(Literal l){
      List<Justification> res = justifies.get(l);

      // If the entry does not exist, make a new one
      if (res==null){
         res = new LinkedList<Justification>();
         justifies.put(l, res);
      }

      return res;
   }
}
```

Bibliography

[1] Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[2] Renata Vieira Álvaro F. Moreira and Rafael H. Bordini. Extending the Operational Semantics of a BDI Agent-Oriented Programming Language for Introducing Speech-Act Based Communication. 2004.

[3] Jomi F. Hübner Mark Jago Natasha Alechina, Rafael H. Bordini and Brian Logan. Automating Belief Revision for AgentSpeak. 2006.

[4] Mark Jago Natasha Alechina and Brian Logan. Resource-Bounded Belief Revision and Contraction. 2006.

[5] Graham Priest and Koji Tanaka. Paraconsistent Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2009 edition, 2009. `http://plato.stanford.edu/entries/logic-paraconsistent/`.

[6] Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Appproach*. Pearson, 2010.

[7] Johannes S. Spurkeland. Using Paraconsistent Logics in Knowledge-Based Systems. DTU Informatics, 2010. BSc Thesis.

[8] Jørgen Villadsen. A paraconsistent higher order logic. In *Paraconsistent Computational Logic*, pages 33–49, 2002. Available at `http://arxiv.org/abs/cs.LO/0207088`.