Hindawi Publishing Corporation
ISRN Artificial Intelligence
Volume 2013, Article ID 632319, 11 pages
http://dx.doi.org/10.1155/2013/632319

Research Article

Belief Revision in the GOAL Agent Programming Language

Johannes Svante Spurkeland, Andreas Schmidt Jensen, and Jørgen Villadsen

Algorithms, Logic and Graphs Section, Department of Applied Mathematics and Computer Science, Technical University of Denmark, Matematiktorvet, Building 303B, 2800 Kongens Lyngby, Denmark

Correspondence should be addressed to Jørgen Villadsen; jovi@dtu.dk

Received 31 May 2013; Accepted 15 July 2013

Academic Editors: C. Kotropoulos, L. Mikhailov, and B. Schuller

Agents in a multiagent system may in many cases find themselves in situations where inconsistencies arise. In order to properly deal with these, a good belief revision procedure is required. This paper illustrates the usefulness of such a procedure: a certain belief revision algorithm is considered in order to deal with inconsistencies and, particularly, the issue of inconsistencies, and belief revision is examined in relation to the GOAL agent programming language.

1. Introduction

When designing artificial intelligence, it is desirable to mimic the human way of reasoning as closely as possible to obtain a realistic intelligence albeit still artificial. This includes the ability to not only construct a plan for solving a given problem but also to be able to adapt the plan or discard it in favor of a new. In these situations the environment in which the agents act should be considered as dynamic and complicated as the world it is representing. This will lead to situations where an agent's beliefs may be *inconsistent* and need to be revised. Therefore, an important issue in the subject of modern artificial intelligence is that of *belief revision*.

This paper presents an algorithm for belief revision proposed in [1] and shows some examples of situations where belief revision is desired in order to avoid inconsistencies in an agent's knowledge base. The agent programming language GOAL will be introduced and belief revision will be discussed in this context. Finally, the belief revision algorithm used in this paper will be compared to other approaches dealing with inconsistency.

2. Motivation

In many situations, assumptions are made in order to optimize and simplify an artificial intelligent system. This often leads to solutions which are elegant and planning can be done without too many complications. However, such systems tend to be more difficult to realize in the real world—simply because the assumptions made are too restrictive to model the complex real world.

The first thing to notice when modeling intelligence is that human thoughts are themselves inconsistent as considered in [2]. It also considers an example of an expert system from [3], where the classical logical representation of the experts' statements leads to inconsistency when attempting to reason with it. From this, one can realize how experts of a field not necessarily agree with one another and in order to properly reason with their statements inconsistencies need to be taken into account. This is an example where it is not possible to uniquely define the cause and effect in the real world. The experts might all have the best intentions but this might not be the case with entities interaction with an agent. Malicious entities may want to mislead the agent or otherwise provide false information which again may lead to inconsistencies. One example of this is when using multiagent systems to model computer security; agents may represent hackers or other attackers on the system in question.

Another important fact about the real world is that it is constantly changing. Agents which find themselves in changing environments need to be able to adapt to such changes; the changes may not lead to inconsistencies. When programming multiagent systems, it might not always be possible or it might be too overwhelming to foresee all consequences and outcomes that the environment provides.

In this paper, a small example will now be presented for the purpose of illustration. The example is inspired by [4, 5], where an agent-based transport information system is considered in order to improve the parking spot problem. That is, having an agent represent each car, they can communicate and coordinate to find an available parking spot nearby. In this paper, cars are considered as autonomous entities which drive their passengers around the city. Each car thus poses as an agent. Such an agent may have the following trivial rules for how to behave in traffic lights,

$$\mathit{green}(X) \longrightarrow \mathit{go}(X) \tag{1}$$

$$\mathit{red}(X) \longrightarrow \mathit{stop}(X) \tag{2}$$

Furthermore, the agent may have a rule specifying that if the brakes of a car malfunction, it cannot stop:

$$\mathit{failing}(\mathit{brakes}, X) \longrightarrow \neg\mathit{stop}(X). \tag{3}$$

Imagine now the situation where the light is *perceived* as green and a car in the crossing lane sends to the agent that its brakes fail. That is, the agent now also believes the following:

$$\mathit{green}(\mathit{me}) \tag{4}$$

$$\mathit{red}(\mathit{other}) \tag{5}$$

$$\mathit{failing}(\mathit{brakes}, \mathit{other}) \tag{6}$$

This situation will now lead to inconsistencies when the agent attempts to reason with its beliefs. From (2) and (5), it is straightforward to deduce $\mathit{stop}(\mathit{other})$, whereas from (3) and (6), $\neg\mathit{stop}(\mathit{other})$ is deduced. Furthermore, by adding rules for the mutual exclusion of the *go* and *stop* predicates and having the rule $\mathit{go}(\mathit{other}) \rightarrow \mathit{stop}(\mathit{me})$ saying to stop if the other does not, the agent will also be able to deduce that it both should go and should not go. The obvious choice for the agent here is to discard the thought of going onwards and stop to let the other car pass. Notice that the exclusion of *go* and *stop* might be achieved by simply using $\neg\mathit{go}$ instead of *stop* or vice versa. Depending on the interpretation of the two predicates, one might want to distinguish the two though—saying that failing brakes of a car means it should *go* seems wrong if it never started.

Assume that the agent makes the right choice and escapes the car crash. The passenger of the car wants to go shopping and the agent thus needs to find an available parking lot. To represent that the agent wants to get to the destination of the shop, it has the following:

$$\mathit{dest}(\mathit{shop}_1) \tag{7}$$

The agent currently does not know the whereabouts of the shop that the passenger wants to go to; so it broadcasts a request for such information. The agent receives a response,

$$\mathit{dest}(\mathit{shop}_1) \longrightarrow \mathit{goto}(\mathit{lot}_1) \tag{8}$$

This basically tells the agent that if it wants to reach shop_1 it needs to get to parking lot_1. This is straightforward; however, shortly after the agent receives a second response from a third agent:

$$\mathit{dest}(\mathit{shop}_1) \longrightarrow \neg\mathit{goto}(\mathit{lot}_1) \tag{9}$$

$$\mathit{dest}(\mathit{shop}_1) \longrightarrow \mathit{goto}(\mathit{lot}_2) \tag{10}$$

The third agent has experienced that the parking lot_1 is full which makes it send the first rule. It then also sends the second rule as a *plan* for getting parked desirably and thereby enabling the passenger to reach the destination shop.

Blissfully adding both responses to the belief base will, however, lead to inconsistencies again. Obviously, (7) can be used with (8) and (9) to obtain $\mathit{goto}(\mathit{lot}_1)$ and $\neg\mathit{goto}(\mathit{lot}_1)$, respectively. Since the agent does not currently have any more information about the two, it does not know which of them to trust.

For more examples, refer to [6]. They are of a more theoretical nature, but may illustrate how one can deal with inconsistencies efficiently. The examples are explained in relation to a tool which was developed by the author and which lets one apply a belief revision algorithm on formulas given to the program. The algorithm will be considered in more details in the next section.

It should be noted that in the above example, one might discuss several ways of trying to fix the problem. For instance, one might suggest that the problem is that the rules should not be strict, and therefore one could introduce a deontic operator meaning "should." This is not considered in this paper—instead the problem considered is that different rules may end up concluding contradictory information (and as such there is negation in actions and beliefs). This is only really a problem for larger and more complex systems, since one may find situation-specific ways of fixing the problem for smaller systems while avoiding a general solution. The example might be fixed by refining the rules and similarly for the expert system presented in [3]. However, consider if the expert system had a huge amount of statements or if all of the huge complexity of the real world had to be considered in the car example, then it might not be as easy to manually find and fix all the sources of inconsistencies. The example is simple but still provides some nice points and illustrates the main problem considered.

Notice that the example above is presented in a form of first-order logic. That is, the exact relation to the actions has not been given. Considering firing rules as adding beliefs, the agent adds information that it believes that it can for example, *go*. This is kind of metareasoning about the actions. Depending on the language, it might be more natural to have the rules execute actions directly. This way firing rules may execute actions instead of adding beliefs.

3. Belief Revision Algorithm

The algorithm considered in this paper is the one proposed by [1] (and [6]). It is based on a combination of the two main approaches to belief revision introduced in [7, 8] and is

```
for all $j = (B, s) \in dependencies(A)$ do
    remove($j$)
end for
for all $j = (A, s) \in justifications(A)$ do
    if $s = []$ then
        $remove(j)$
    else
        $contract(w(s))$
    end if
end for
$remove(A)$
```

Algorithm 1: Contraction by belief A.

$$\begin{tikzcd}
((\boxed{1}), []) \arrow[d] & ((\boxed{4}), []) \arrow[d] & ((\boxed{2}), []) \arrow[d] & ((\boxed{5}), []) \arrow[d] & ((\boxed{3}), []) \arrow[d] & ((\boxed{6}), []) \arrow[d] \\
(\boxed{1}) \arrow[d] & (\boxed{4}) \arrow[dl] & (\boxed{2}) \arrow[d] & (\boxed{5}) \arrow[dl] & (\boxed{3}) \arrow[d] & (\boxed{6}) \arrow[dl] \\
(go(me), [(\boxed{4}), (\boxed{1})]) \arrow[d] & & (stop(other), [(\boxed{5}), (\boxed{2})]) \arrow[d] & & (\neg stop(other), [(\boxed{6}), (\boxed{3})]) \arrow[d] & \\
go(me) & & stop(other) & & \neg stop(other) &
\end{tikzcd}$$

Figure 1: Graph over the beliefs and justifications in the first part of the example from Section 2.

extending work done in, for example, [9, 10] by treating rules and facts the same.

The basic principle of the algorithm is to keep track of what beliefs the agent has and how the agent may *justify* these beliefs. The idea is based on the human reasoning process: if two contradictory beliefs arise, one of them is selected and the other is discarded. Of course, one must also discard whatever beliefs may end up concluding the discarded belief as they can no longer be trusted either. This process of discarding beliefs is referred to as *contraction* by beliefs. This will be made more concrete in the following.

The agent is defined as having beliefs consisting of ground literals, l or $\neg l$, and rules. The rules take the form of universally quantified positive Horn clauses, $l_1 \wedge l_2 \wedge \cdots \wedge l_n \rightarrow l$ and the agent is required to reason with a weak logic, W, which only has generalized modus ponens as inference rule (i.e., if the antecedent of an implication holds, the consequent can be inferred). The inference rule is formally as follows, where δ is a substitution function replacing all variables with constants:

$$\frac{\delta(l_1), \ldots, \delta(l_n) \quad l_1 \wedge \cdots \wedge l_n \longrightarrow l}{\delta(l)} \tag{11}$$

The approach is now to detect inconsistencies in the belief base. Notice that this is a simple rule, $l \wedge \neg l \rightarrow \neg consistent(l)$. If an inconsistency is detected, the least preferred belief is removed. Furthermore, the rules from which the belief can be derived are removed along with the beliefs, which can only be derived from the removed belief. This assumes two things. First, that we keep track of how the beliefs relate to each other, and second, that there is a measure of how preferred a belief is.

To deal with the first problem, the notion of justifications is used. A justification is a pair, (A, s), of a belief A and a support list s. The support list contains the rule used to derive A together with all the premises used for firing that rule. This means that the percepts and initial knowledge will have an empty support list. The justifications for $green(me)$ and $\neg stop(other)$ from the example in Section 2 are then as follows:

$$\begin{gathered}(green(me), []) \\ (\neg stop(other), [failing(brakes, other), \\ failing(brakes, X) \longrightarrow \neg stop(X)])\end{gathered} \tag{12}$$

Notice that a belief may have several justifications. If the light had been green for the other and there was an exclusivity rule $go(X) \rightarrow \neg stop(X)$, then $\neg stop(other)$ would also have the justification:

$$(\neg stop(other), [go(other), go(X) \longrightarrow \neg stop(X)]) \tag{13}$$

Having this data structure, the beliefs and justifications can be regarded as a directed graph: incoming edges from the justifications of a belief and outgoing edges to justifications containing the belief in its support list. Figure 1 shows such a graph for the first part of the presented example. The elements at an odd depth are justifications, whereas elements of an even depth are beliefs. To make the graph fit, references to the formulas have been used instead of the actual formula where possible. One may also notice that the agent has the two contradictory nodes, which means that either of the two subgraphs holding one of the two nodes must be contracted.

In [9, 10], the successors of a belief (justifications with the belief in the support list) are denoted dependencies. The algorithm is then simply considered to have a list of justifications and dependencies for each belief and recursively traverses the elements of these two lists to remove beliefs which are justifying or justified only by the belief selected for contraction. More specifically, Algorithm 1 provides pseudo code for the contraction algorithm similar to that from [9, 10]. The contraction algorithm assumes that an inconsistency has

```
Knowledge {
    % Mutual exclusion rules of go and stop predicates.
    neg (stop (X)) :- go (X).
    neg (go (X)) :- stop (X).
}
```

Algorithm 2

been detected and the least preferred belief is given as input. The $w(x)$ function takes a support list as input and returns the least preferred belief of that list.

It is worth noting that the belief base needs to be in the quiescent setting in order for the algorithm to work properly. Basically what this means is that all the rules which may add new information must have been fired before executing the algorithm. This is due to that if more beliefs can be inferred, the belief for contraction might be inferred again by other beliefs than the ones removed by the algorithm. Consider, for example, the belief base consisting of $\{A \rightarrow C, B \rightarrow C, A, B, \neg C\}$. Then firing the first rule adds a contradiction and if contracting on C without firing the second rule then C may again be derived, introducing the inconsistency again. Notice, however, that this requirement is only for the algorithm to work optimally. Not firing the second rule does not make the algorithm work incorrectly but simply does so that it cannot be guaranteed that the contracted belief cannot be inferred from the current beliefs again after contraction.

It may be desirable for the algorithm to also contract beliefs which no longer have justifications because they have been removed in the contraction process, cf. [9, 10]. Even though keeping the beliefs in the belief base might not currently lead to inconsistencies, something is intuitively wrong with keeping beliefs which are derived from what has been labeled wrong or untrustworthy information.

The problem of measuring how preferred a belief is does not have a trivial solution. In fact, the problem is passed on to the designer of the multiagent system in question. The requirement from the perspective of the algorithm is that there is an ordering relation such that for any two beliefs it is decidable which one is more preferable to the other. However, in [9] an algorithm is presented for preference computation.

Consider again the example from Section 2. Intuitively, percepts of the agent should be of highest preference since the agent ought to trust what it itself sees. However, in the example, this would lead to the contraction of the belief $\neg stop(other)$ which will mean that the agent will decide to move forward and crash with the other car. Instead here the belief received from the other agent should actually have a higher preference than the percepts. This is, however, in general, not desirable as other agents might not be trustworthy. This is even though the other agents might have good intentions. Consider, for example, the plan exchange between the agents in the example. Here, the first agent sends a plan for getting to the closest parking lot not knowing it is full. If the agent chose to follow this plan and on the way perceived a sign showing the number of free spots in the parking lot, this percept should be of higher preference than following the plan through. This illustrates the care which needs to be taken in the process of deciding upon nonpreferred beliefs.

4. GOAL

GOAL is a language for programming rational agents and as such is a target of interest in relation to belief revision. This section will examine GOAL and how it conforms to belief revision of inconsistent information. First, the basic concepts of GOAL are explained and afterwards belief revision is considered in GOAL. This paper will not explain how to set up and use GOAL, instead the reader is referred to [11].

4.1. The Basics. The basic structure of a GOAL multiagent system consists of an environment and the agents which interact within this environment. The agents are according to [12] connected to the environment by means of entities. That is, an agent is considered to be the mind of the physical entity it is connected to in the multiagent system. Agents need not be connected to an entity though; however, they cannot interact with the environment if they are not. Having agents that are not part of the environment can be useful in, for example, organizational centralized multiagent systems, where there may be an agent whose only task is to coordinate the organization.

Agents may consist of the following components:

(1) knowledge;

(2) beliefs;

(3) goals;

(4) module sections;

(5) action specification.

Knowledge is what is known to be true. It should be considered as axioms and as such should not be allowed to be inconsistent. Beliefs, however, represent what the agent thinks is true and may be both false and ambiguous. That is, different rules may lead to inconsistent beliefs. The representation of knowledge, beliefs, and goals is all in a Prolog based manner by standard (see [12] for more details). While it may be arguably very few things in the real world that are certain enough to be declared axioms, the mutual exclusion of the *go* and *stop* predicates should be. Since the two predicates are direct opposites, it does not make sense to have both, for example, $go(me)$ and $stop(me)$. Declaring these rules as initial knowledge may look as Algorithm 2.

```
beliefs {
    % Green and red light rules.
    go (X) :- green (X).
    stop (X) :- red (X).
}
```

Algorithm 3

```
actionspec {
  /* Action for the car agent to drive towards Dest provided it is desired and
        that it can go onwards. */
  drive (Dest) {
     Pre {dest (Dest), me (Me), go (Me)}
     post {true}
  }
}
```

Algorithm 4

The two *neg* predicates represent strong negation and will be explained in Section 4.2.

One might argue that the rules for green and red lights are accepted worldwide and thus should be considered axioms. However, global consensus and being a universal truth are not the same. First, the trend may change, and second, the agent may find itself in unforeseen situations in which such a rule cannot apply. Consider the example where green light should not allow for *go* since the other car cannot fulfill the rule of the red light. Another example is that of [1], where there is global consensus that birds fly and yet penguins, which are birds, cannot fly. These are both examples of why one cannot assume not to face inconsistencies when exposing multiagent systems to the real world. We therefore choose to declare the two traffic light rules as beliefs rather than knowledge as shown in Algorithm 3.

The action specification follows a STRIPS-style specification (see e.g., [13, ch. 10]) and actions are defined using a name, parameters, preconditions, and postconditions. Imagine that the car agent has a drive action which represents the agent driving to a desired destination. The action takes the destination to drive to as parameter, and has as preconditions that it must desire to get to the destination and that it can go onwards. The *me* predicate is a built-in predicate for the agent to obtain its own ID and the action specification may now look as Algorithm 4.

The postcondition may seem to be somewhat odd—driving somewhere ought to change the agent's state. This is due to thinking of the drive action as a *durative* action. In GOAL, there is a distinction between the two types of actions: *instantaneous* and *durative* [12]. Durative actions take time to perform, whereas instant actions to will finish immediately and thereby affect the system immediately. Therefore, the approach seems to be to let the outcome of durative actions be a consequence of their effect on the environment. That way durative actions can also be interrupted and their effect might not (entirely) come through. Thus, the drive action is durative, since the agent will not instantly reach its destination. Furthermore, the environment can send percepts to the agent (such as a red light), which will interrupt the action (if the agent chooses to react to the percept). When the agent perceives that the lights become green, it may resume driving.

If one wants to avoid the kind of metareasoning discussed in Section 2, the premise of the green rule might have been used directly when defining the requirements of the drive action. That is, instead of the $go(Me)$ precondition, one might use $green(Me)$. Notice, however, that this requires instantiation of the premise and makes the rule less general. This also means that the rule cannot be used in other contexts if desired (unless again explicitly defined). Basically, these considerations depend on how one wants to design the system. Yet another approach would be to use the premise in the motivation for taking the action (see the part about the main module below); however, this implies the same considerations.

The environment is specified by means of an environment interface standard; for more information about this standard [12], refer to, for example, [14]. It is beyond the scope of this paper to uncover how to program environments for GOAL programs; however, we assume that there is an environment in which the agents can perceive the traffic lights. The agent has a percept base which contains the percepts of the current reasoning cycle. It is up to the programmer to make the agent capable of reacting to such percepts. This can be done by using the predicate *percept*, which is a predicate holding the current percepts of the agent.

The module sections define the agent's behavior. There are two modules by standard: the event module and the main module. The purpose of the main module is to define how the agent should decide what actions to perform, whereas the purpose of the event module is to handle events such as percepts and incoming messages so that the belief base is always up to date when deciding upon actions cf. [12].

```
event module {
  program {
    % Green and red light percepts - on.
    forall bel (percept (green (X)), not (green (X))) do insert (green (X)).
    forall bel (percept (red (X)), not (red (X))) do insert (red (X)).

    % Green and red light percepts - off.
    forall bel (green (X), percept (not (green (X)))) do delete (green (X)).
    forall bel (red (X), percept (not (red (X)))) do delete (red (X)).
  }
}
```

Algorithm 5

```
main module {
  program {
    % Drive economically if low on gas.
    if bel (low (gas), dest (X)) then drive (dest (X), eco).

    % Drive fast if busy.
    if bel (busy, dest (X)) then drive (dest (X), fast).

    % Drive comfortably otherwise.
    if bel (dest (X)) then drive (dest (X), comfort).

    % If no driving options the car simply idles.
    if true then skip.
  }
}
```

Algorithm 6

Therefore, the event module is always the first to run in an agent's reasoning cycle. To add the logic for actually making the agent believe the light perceptions, the event module of Algorithm 5 may be used where *insert* and *delete* are two built-in functions for adding and deleting beliefs, respectively.

The *bel* predicate is a built-in predicate to indicate that the agent believes the argument(s) of the predicate.

The main module is not that interesting when the agent does not have more actions defined since there are not really any strategic choices to regard in relation to the choice of action. It might consider whether or not to drive or brake. Depending on how the environment is defined, braking might be implicitly assumed by saying that if the car is not driving, it has stopped. Instead of considering a brake action, here is considered a more sophisticated *drive* action. One may assume that the *drive* action takes an extra argument which defines the mode that the car should drive in. Assuming that when low on gas it drives economically, and if busy it drives fast; the main module of Algorithm 6 may define the choice of actions.

Notice that the precondition stating that the *drive* action must have a destination is actually obsolete now since this is ensured when deciding upon action in the main module. The main module may evaluate actions in different kinds of orders. The default is linear which means that, for example, the economical option is chosen over the others if applicable. The *skip* action is not built-in but may be defined by simply having *true* as pre- and postconditions.

Agents communicate using a mailbox system. Mails are handled similarly to percepts, with the main difference that the mailbox is not emptied in every reasoning cycle. GOAL supports three moods of messages: indicative, declarative, and interrogative. These three moods basically represent a statement, a request and a question and are denoted using an operator in front of the message. Similar to the *percept* predicate, there is a *received* predicate over received messages. This predicate takes two arguments: the agent who sent the message and the message itself. Messages must be a conjunction of positive literals. Therefore, handling the message that the other agent's brakes fail can be done by adding Algorithm 7 to the program shown in Algorithm 5 in the event module where : is being the indicative operator.

By requiring messages to be a conjunction of positive literals, agents cannot share rules or plans. It is simply not possible to send or receive the rules (8), (9), and (10) from the second part of the example. This also means that the rule base of an agent will be static and is designed *offline*. Thus, the graph of Algorithm 1 will be less complex during runtime.

Another lack of expressiveness in GOAL is that of representing inconsistencies. This will be examined further in the next section.

```
% Add the belief received from another agent that its brakes fail.
if bel (received (A,:failing (brakes, A))) then insert (failing (brakes, A))
  + delete (received (A,:failing (brakes, A))).
```

Algorithm 7

4.2. Inconsistencies. While having the tools for implementing the algorithm dealing with inconsistencies, a rather crucial point is to be noted. Since the knowledge representation language of the belief base in GOAL is Prolog, the rules will all take the form of positive Horn clauses with a positive consequent. This means only positive literals can be concluded using the rules in the belief base. Furthermore, the action specifications ensure that if a negative literal is added then it is not actually added to the belief base. Instead, if the positive literal is in the belief base it is removed and otherwise nothing happens. This is due to the closed world assumption. This means that an agent will never be able to represent both the positive and the negative of a literal in its belief base. That is, GOAL simply does not allow for representing inconsistencies when using Prolog as knowledge representation language.

One of the advantages of GOAL is that its structure allows for selecting different knowledge representation languages depending on what is best with regard to modeling the system at hand. At current stage, it is only Prolog which has been implemented as knowledge representation language; however, work is done on implementing answer set programming. This allows for both negation as failure as in Prolog but also for a notion of strong negation (see, e.g., [15] for more information about answer set programming) which allows for representing incomplete information. This means that using answer set programming as knowledge representation language would be quite powerful when dealing with systems in which inconsistencies might arise. It is not implemented yet though; so in the following, strong negation will be discussed in relation to GOAL using Prolog.

In order to allow for the representation of inconsistencies, the notion of strong negation is introduced in Prolog. In [16], this kind of negation does not rely on the closed world assumption. It explicitly says that the negation of a formula succeeds whereas negation as failure says that the formula does not succeed, but also that it does not explicitly fail either. In other words, negation as failure can be read as "it is not currently believed that."

The basic principle is now to consider the strong negation, $\neg$, of a predicate as a positive literal. That is, for literal, p, p' can be regarded as a positive literal with the meaning $\neg p$. In terms of Prolog, this means querying p will succeed if p holds, fail if $\neg p$ holds, and be inconsistent if both holds. In [16], it is furthermore considered possible to return the value unknown to such queries if neither p nor $\neg p$ can be proven in the current Prolog program. Another interesting observation they made is that the closed world assumption can be defined for any literal in the following way (where *not* denotes negation as failure):

$$\neg p(X_1, \ldots, X_n) \longleftarrow not\,(p(X_1, \ldots, X_n)) \qquad (14)$$

The interested reader might also see [17] in relation to strong negation and logic programming.

In the terms of GOAL, this means that it is fairly straightforward to introduce the notion of strong negation, and the *neg* predicate introduced in Section 4.1 serves exactly this purpose. There is no explicit problem in having both the belief and its negation in the belief base: it is required in the event module to check if the belief base is still consistent by querying whether or not $neg(X), X$ follows from the belief base and act accordingly. It may be necessary to introduce a *pos* predicate denoting positive literals as GOAL does not allow for the query $bel(X)$.

4.3. Belief Revision. This section will consider how to implement the belief revision algorithm described in Section 3 in GOAL.

The event module is the first thing that is executed in an agent's reasoning cycle and since it is desirable to have an up-to-date belief base when performing belief revision, the belief revision algorithm should be implemented as the last procedure in the event module.

GOAL relies on Prolog as representation language (in most cases), which conforms well to the weak logic defined for use with the belief revision algorithm in [1] since Prolog programs consist of Horn clauses and literals.

The question is then how to associate and represent the justification and dependency lists. One idea is to simply let them be beliefs of the agent. This way one just has to make sure to add a justification when adding a belief. The rules for adding percepts to the agent from Section 4.1 may be extended to also constructing a justification as shown in Algorithm 8.

Similarly, the justification is deleted when the percept is no longer valid, as shown above. While the case is rather trivial when dealing with percepts (as they do not have any justifications), a similar approach may be taken for the rules, the actions, and when adding beliefs from messages of other agents. We need to consider the actions carefully, since the motivation for executing an action is specified in the main module, whereas the postconditions (i.e., effect) of the actions are specified in the action specification. Each effect of an action should be supported by a conjunction of the motivation for taking the action and the preconditions of the action. There are several ways for obtaining this

```
% Add beliefs and justifications from red light percepts.
forall bel (percept (red (X)), not (red (X))) do insert (red (X))
  + insert (just (red (X), [], p)).
% Remove belief and justification when no longer red light.
forall bel (red (X), percept (not (red (X)))) do delete (red (X))
  + delete (just (red (X), [], _)).
```

Algorithm 8

```
actionspec {
% Action for the car agent to drive which also adds justifications.
   drive (Dest, Pre) {
     pre {dest (Dest), me (Me), go (Me), append (Pre, [dest (Dest), go (Me),
           action (drive)], S)}
     post {at (Dest), just (at (Dest), S)}
   }
}
```

Algorithm 9

conjunction. A simple one is to simply let the action take them as parameters. The provided action is instantaneous; the agent is immediately at its destination; so the action specification could look like Algorithm 9.

The idea is to include the motivation from the main module as an argument to the action and then append it to a list of preconditions of the action to obtain *s*. The *action* predicate then corresponds to the rule used for deriving the belief. In this case it is a plan, which has been executed. Since actions and main modules cannot be altered dynamically, another predicate might be added as precondition, for example, *not contracted*(*action*(*drive*)). If an action or particular instance of an action is then contracted using the belief revision algorithm, for example, the belief *contracted*(*action*(*drive*)) may simply be added to invalidate any future run of that action. If the agent finds reason to believe in the action again, it may simply remove the belief again.

The case of durative actions is more difficult to handle. The effects of a durative action appear as changes in the environment, which the agent will then perceive. However, since percepts create a justification with an empty support list, there is no way of telling whether the percept is from a change in the environment due to an action that the agent has performed or simply due to the environment itself (or even other agents interacting in the environment). In other words, durative actions cannot be contracted. One might attempt to solve this problem by implementing the environment such that it provides justifications for changes happening as a result of actions and then keep track of how these justifications relate to the percepts. However, it seems wrong to let the environment handle part of the agent's reasoning and it might couple the agent and the environment too much.

One might raise the questions why include actions in the contraction process and what does it actually mean to contract an action? Actions may be reasons for adding new beliefs (e.g., changes in the environment as a result of an action). The agent will need some way of justifying these beliefs, especially because a rule might trigger an action, which then adds a belief that renders the belief base inconsistent. If there is no justification; for the action, then it is not possible to trace back to the rule originally triggering the action leading to the inconsistency. Furthermore, it might be that some actions simply lead to undesirable outcomes and therefore the agent should not want to do them again. Therefore, one might decide to contract them and thereby disable executing them in the future. Since actions are not deleted, they may be enabled again in the future if desirable.

We have provided means for representing justifications, so the next step is to check for inconsistencies, which, as argued above, will be done as the last thing in the event module. Then if two contradictory beliefs are found, the less preferred of the two is marked for contraction.

The contraction itself then happens by three blocks. In the first all the positive beliefs marked for contraction are contracted and in the second all the negative. These two blocks follow contraction Algorithm 1 with the exception of the recursive call. This is what the third block is for. The recursive call is emulated by marking all the least preferred elements of the support lists to contract and in the third block the first of the recursive calls are then performed on these. Again, the recursive call is emulated by marking beliefs for contraction. However, since the program is executed sequentially, it has now passed the three blocks for contraction. The idea is then to prohibit the agent from performing any actions until there are no beliefs marked for contraction. Then the agent will

```
% Detect inconsistencies
#define poscontract (X) bel (p (neg (X), Pn), p (X, Pp), Pn > Pp).
#define negcontract (X) bel (p (neg (X), Pn), p (X, Pp), Pn < Pp).
% Contract all least preferred positive beliefs
listall C <- poscontract (X) do {
  forall just (Y, S), member (Z, C), member (Z, S) do delete (just (Y, S)).
  forall just (Y, []), member (Y, C) do delete (just (Y, _)).
  forall member (Y, C), just (Y, S) bel (w (just (Y, S), Z)) do insert (contract (Z)).
  forall member (Y, C) do delete (Y).
}
% Contract all least preferred negative beliefs
listall C <- negcontract (X) do {
  forall just (Y, S), member (Z, C), member (Z, S) do delete (just (Y, s)).
  forall just (Y, []), member (Y, C) do delete (just (Y, _)).
  forall member (Y, C), just (Y, S) bel (w (just (Y, S), Z)) do insert (contract (Z)).
  forall member (Y, C) do delete (Y).
}
% Recursive contraction
listall C <- contract (X) do {
  forall just (Y, S), member (Z, C), member (Z, S) do delete (just (Y, S)).
  forall just (Y, []), member (Y, C) do delete (just (Y, _)).
  forall member (Y, C), just (Y, S) bel (w (just (Y, S), Z)) do insert (contract (Z)).
  forall member (Y, C) do delete (Y).
}
```

Algorithm 10

do nothing and the next reasoning cycle will start. This time it will go directly to the third cycle and continue emulating recursive calls by doing this until no more beliefs are marked for contraction and the agent is allowed to perform actions again (Algorithm 10).

However, this solution is not optimal, since it lets the agent idle for several cycles while it is revising its thoughts. Though the number of recursive calls is most likely quite low because of the simplified graph due to static rules as mentioned in Section 4.1, it would be better to have a Prolog contraction procedure, which can make use of recursion. One might, for example, import such a procedure in the knowledge section such that it will mark all the beliefs for contraction recursively and in the next cycle actually do the removal of them. This way only one extra cycle is used for contracting beliefs.

Another possibility is to recursively contract beliefs in the event module. This is done by moving the contraction algorithm into a separate module, which is then imported in the event module (see, e.g., [11] for how to import) and can be called recursively within itself. This would obviously be the most efficient solution as it only takes the calculation time and no reasoning cycles.

Until now, the implementation of the preference relations has not been discussed. In the above code, it is assumed that a preference of a belief is added as a predicate cf. justification. Furthermore, it is assumed that when adding a nonempty support list of a justification, a predicate w, which specifies the least preferred belief in a support list, is added to the belief base. This simplifies the least preferred belief queries; however, one should keep in mind that these predicates all should be deleted when also deleting the corresponding belief.

In [9, 10], the algorithm was considered with regards to an implementation in Jason, and they argued that the quiescent setting could not be guaranteed. In our implementation, an action may not be activated for a long time, but it may still lead to inconsistencies. Therefore, the same argument can be made in our case. However, when querying the belief base for inconsistencies (which is done every reasoning cycle), the Prolog engine will attempt to evaluate all the rules in the belief base in order to search for a proof. This means that the quiescent setting is guaranteed for the belief base, but not for the action rules.

5. Other Approaches

Previous work, [2], considered a four-valued logic proposed by [3] in order to deal with inconsistent information. The main difference between this approach and the algorithm considered here is that the algorithm attempts to recover the belief base from an inconsistent state to a consistent state. It does so by attempting to get rid of the information which was the cause of the inconsistency. The four-valued logic on the other hand attempts to reason with the inconsistent knowledge base while actually preserving the inconsistency in the system.

At first glance, it seems to be more desirable to recover the belief base from an inconsistent state to a consistent state.

However, just like determining the preferences of the beliefs, this also has some complications. For instance, if the agent makes the right choice in the example from Section 2, it will contract the rules for the light signal. This means that when the danger has passed, then the agent will not know to go when it is green light and stop when it is red light. That is, deleting information from the knowledge base might not always be the best choice as the agent might actually delete some vital beliefs. The problem is that the beliefs might hold in most situations where they should be applied but in some more rare cases they may lead to inconsistencies.

Handling inconsistencies is still a debated topic and there is no full solution yet. One might take several different approaches for dealing with the above problem. One is to do as with the actions where one simply disables rules instead of completely deleting them. Then they may be reconsidered later. One could also attempt to combine the four-valued approach with the contraction algorithm. That is, use contraction and if a requirement arises that a rule which is known to lead to inconsistencies needs to be used again, one can attempt to use the paraconsistent logic to reason with this rule. Yet another approach is rule refinement instead of rule contraction. If the agent, for example, detects the cause of the inconsistency, it might be able to repair or refine the rules in question. If, for instance the car agent realizes that the cause of the problem is the failing brakes with regard to the red light rule, it could refine it into the following rule:

$$red(X) \wedge not \;\; failing(brakes, X) \longrightarrow stop(X) \tag{15}$$

Even though it might seem to be the smartest solution, this is not guaranteed to always have a solution and finding such a solution might prove very complicated.

6. Conclusion

It has been argued why belief revision is an important issue. A particular algorithm for belief revision has been considered. It has the advantages of being efficient and straightforward to implement; however, it has the disadvantages that it is only defined for a weak logic of the agent and that it requires the nontrivial question of a preference ordering. Issues of such an ordering have been pointed out. Furthermore, issues with deleting information, which may be important in many cases even though it is inconsistent, have been pointed out.

GOAL has been examined in relation to belief revision. It has the strengths of using logic programming which means that it is very easy to learn for people with a background in logic programming and it provides elegant logical solutions. Furthermore, the restrictions of logic programming conform well to the restrictions of the weak logic of the belief revision algorithm. However, it has been identified that at current state, the GOAL language is actually more restrictive than required for the algorithm which results in that inconsistencies cannot be represented at all. The introduction of strong negation has been considered in order to mitigate this problem. Furthermore, using answer set programming as knowledge representation language may also deal with this problem when it is supported by GOAL. Another less critical issue with GOAL is that it does not provide the means for plan sharing.

Acknowledgments

The authors would like to thank Koen V. Hindriks and M. Birna van Riemsdijk for their comments.

References

[1] H. H. Nguyen, “Belief revision in a fact-rule agents belief base,” in *Agent and Multi-Agent Systems: Technologies and Applications*, A. Håkansson, N. T. Nguyen, L. Ronald Hartung, R. J. Howlett, and L. C. Jain, Eds., vol. 5559 of *Lecture Notes in Computer Science*, pp. 120–130, Springer, Berlin, Germany, 2009.

[2] J. S. Spurkeland, *Using paraconsistent logics in knowledge-based systems [B.Sc. thesis]*, Technical University of Denmark, 2010.

[3] J. Villadsen, “Paraconsistent knowledge bases and many-valued logic,” in *Proceedings of the International Baltic Conference on Databases and Information Systems (Baltic DB&IS ’02)*, H.-M. Haav and A. Kalja, Eds., vol. 2, pp. 77–90, Institute of Cybernetics at Tallinn Technical University, 2002.

[4] N. Bessghaier, M. Zargayouna, and F. Balbo, “An agent-based community to manage urban parking,” in *Advances on Practical Applications of Agents and Multi-Agent Systems*, Y. Demazeau, J. P. Müller, J. M. C. Rodríguez, and J. B. Pérez, Eds., vol. 155 of *Advances in Intelligent and Soft Computing*, pp. 17–22, Springer, Berlin, 2012.

[5] N. Bessghaier, M. Zargayouna, and F. Balbo, “Management of urban parking: an agent-based approach,” in *Artificial Intelligence: Methodology, Systems, and Applications*, A. Ramsay and G. Agre, Eds., vol. 7557 of *Lecture Notes in Computer Science*, pp. 276–285, Springer, Berlin, Germany, 2012.

[6] H. H. Nguyen, *A belief revision system [B.Sc. thesis]*, University of Nottingham, 2009.

[7] C. E. Alchourron, P. Gärdenfors, and D. Makinson, “On the logic of theory change: partial meet contraction and revision functions,” *The Journal of Symbolic Logic*, vol. 50, pp. 510–530, 1985.

[8] J. Doyle, *Truth Maintenance Systems for Problem Solving*, Massachusetts Institute of Technology, Cambridge, Mass, USA, 1978.

[9] N. Alechina, M. Jago, and B. Logan, “Resource-bounded belief revision and contraction,” in *Declarative Agent Languages and Technologies III*, M. Baldoni, U. Endriss, A. Omicini, and P. Torroni, Eds., vol. 3904 of *Lecture Notes in Computer Science*, pp. 141–154, Springer, Berlin, Germany, 2005.

[10] N. Alechina, R. H. Bordini, J. F. Hübner, M. Jago, and B. Logan, “Automating belief revision for AgentSpeak,” in *Declarative Agent Languages and Technologies IV*, M. Baldoni and U. Endriss, Eds., vol. 4327 of *Lecture Notes in Computer Science*, pp. 61–77, Springer, Berlin, Germany, 2006.

[11] K. V. Hindriks and W. Pasman, *GOAL User Manual*, 2012.

[12] K. V. Hindriks, *Programming Rational Agents in GOAL*, 2011.

[13] S. Russell and P. Norvig, *Artificial Intelligence—A Modern Approach*, Pearson Education, 3rd edition, 2010.

[14] T. M. Behrens, K. V. Hindriks, and J. Dix, “Towards an environment interface standard for agent platforms,” *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 4, pp. 261–295, 2011.

[15] V. Lifschitz, *What Is Answer Set Programming*, University of Texas at Austin, 2008.

[16] C. Baral and M. Gelfond, "Logic programming and knowledge representation," *The Journal of Logic Programming*, vol. 19-20, no. 1, pp. 73–148, 1994.

[17] G. Wagner, "Logic programming with strong negation and inexact predicates," in *Vivid Logic*, vol. 764 of *Lecture Notes in Computer Science*, pp. 89–110, Springer, Berlin, Germany, 1994.