Reason Maintenance for Production Systems

L.C. van der Gaag and C. de Koning

UU-CS-1994-13
March 1994

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Reason Maintenance for Production Systems

L.C. van der Gaag and C. de Koning

Technical Report UU-CS-1994-13
March 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Reason Maintenance for Production Systems

Linda van der Gaag
Utrecht University, Department of Computer Science
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Kees de Koning
University of Amsterdam, Department of Social Science Informatics
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands

Abstract

In developing a rule-based bottom-up inference system employing some kind of non-monotonic reasoning, a major part of the programming effort is spent on dynamic maintenance of the validity and consistency of the information the system is working on. Programming this type of maintenance can be rendered unnecessary by supporting the production system with a reason maintenance system: the reason maintenance system then performs the task of ensuring the validity and consistency of information. In this paper, we propose an architecture for supporting a rule-based bottom-up inference system with a justification-based reason maintenance system. The architecture proposed is outlined and the encoding of the information that is exchanged between the two systems co-operating in the architecture is detailed. In addition, we indicate how bottom-up programming benefits from the support provided.

1 Introduction

Rule-based bottom-up inference systems have been developed as early as the 1970s and are still gaining in popularity. These production systems often employ some kind of *non-monotonic reasoning* to allow for dealing with incomplete information. Without complete information on the problem at hand, it may be necessary to start reasoning on the assumption that certain facts are (not) true. If evidence to the contrary is revealed, information derived on this assumption may no longer be valid and has to be reconsidered by the system with respect to its validity. The problem in this type of reasoning is to guarantee at all times the validity and consistency of the information the system is working on.

When developing a rule-based bottom-up inference system employing some kind of non-monotonic reasoning, a programmer has to ensure explicitly that the system itself takes care of the maintenance of the validity and consistency of its information base. To this end, programming languages for building production systems generally provide the programmer with special constructs, such as actions for deleting obsolete information. Experience with bottom-up programming, however, reveals that a major part of the effort is spent on programming this type of maintenance. A means of relieving the programmer of the obligation of programming maintenance is to support a rule-based bottom-up inference system by a *reason maintenance system*, [Smith & Kelleher, 1988]; such a system keeps track of the beliefs of the production system.

In this paper, we propose supporting a rule-based bottom-up inference system by a justification-based reason maintenance system and outline an architecture in which the two systems co-operate. In addition, we show how bottom-up programming benefits from the support provided. In our discussion, we focus on rule-based bottom-up inference systems written in OPS5. The *OPS-family* of programming languages is inextricably connected with bottom-up programming, [Brownston *et al.*, 1985; Neiman & Martin, 1986; Forgy, 1991]. This family of languages already has a long standing history, yet many of today's production systems are still being built using an OPS-language. The success of these languages can be attributed to a large extent to the matching algorithm incorporated, facilitating the development of efficient systems, [Forgy, 1982]. Although our discussion is focused on the OPS5-language, the results reported apply to other members of the OPS-family of programming languages as well.

The paper is organized as follows. In Section 2 we briefly review the OPS5-language and rule-based bottom-up inference systems in general, and provide some preliminaries on reason maintenance. In general, a problem solver and a reason maintenance system co-operate in a highly modular architecture. In Section 3 we discuss which information is to be exchanged between a production system written in OPS5 and a justification-based reason maintenance system, and how this information is encoded; in Section 4 we detail the architecture in which the systems co-operate. Section 5 discusses the advantages that arise from the support proposed. The paper is rounded off with some conclusions and suggestions for further research in Section 6.

2 Preliminaries

In this section, we provide some preliminaries on rule-based bottom-up inference systems and justification-based reason maintenance systems.

2.1 Rule-Based Bottom-Up Inference Systems

A rule-based bottom-up inference system consists of three major components: a *production memory*, a *working memory*, and an *inference engine*. In the production memory, the problem-solving knowledge of the system is stored, represented in IF-THEN rules called *productions*. The working memory may be viewed as a global data space and holds the case-specific data the system is working on. The actual problem solving is performed by the inference engine: it applies the problem-solving knowledge represented in the productions to the case-specific data in the working memory.

The Working Memory

In an OPS5-system, the data in the working memory is represented as statements concerning objects and their associated attribute-value pairs. These statements are stored in so-called *working memory elements*, taking the following form:

$$t\text{:}\,(\langle object\rangle\ \langle attribute_1\rangle\langle value_1\rangle\ \cdots\ \langle attribute_n\rangle\langle value_n\rangle)$$

where $n \geq 0$ and t is a unique integer assigned to the element. The integer t of a working memory element is called its *time-tag* and indicates the moment the element was last operated on during problem solving. The time-tags of the working memory elements are employed by the inference engine.

The Production Memory

The productions in the production memory of a bottom-up inference system are statements of the following form:

(p ⟨*LHS*⟩ --> ⟨*RHS*⟩)

where *LHS* is called the *left-hand side* of the production, and *RHS* is its *right-hand side.* The left-hand side of a production is a sequence of *condition elements*, each of which specifies a test on the contents of the working memory. A condition element may be preceded by a (logical) negation, denoted as -; for technical reasons, such a *negated condition element* may not appear as the first element in the left-hand side of a production.

The right-hand side of a production is composed of *actions.* For the present paper, only actions specifying operations on the working memory are relevant; these are the addition (the action `make`), removal (the action `remove`) or modification (the action `modify`) of a working memory element. A production need not specify any condition elements: it may consist of a single action only. Such a production is called a *top-level instruction.* In both the left-hand side and the right-hand side of a production, variables are allowed to occur; a variable has the production it is specified in for its scope.

A production is interpreted as stating: if the tests specified in its left-hand side succeed, then the accompanying actions in its right-hand side may be performed.

The Inference Engine

Informally speaking, the inference engine of a bottom-up inference system operates in a cyclic fashion, selecting appropriate productions and executing their actions in turn. One of the basic operations performed by the inference engine is to decide whether or not a production's left-hand side succeeds given the current contents of the working memory. In selecting appropriate productions, it tries to *match* the left-hand sides of the productions against the working memory elements in the working memory. To this end, it views a condition element of a production as a partial description of a working memory element. In the case that this partial description 'fits' a specific working memory element, binding occurring variables to constant values if necessary, a match is found for that condition element. The entire left-hand side of a production *succeeds* given the contents of the working memory if the following properties hold:

- every positive condition element matches a working memory element; and
- there are no matching working memory elements for negated condition elements.

For every match found, the inference engine creates a *production instantiation*: a production instantiation consists of a production and a set of working memory elements matching its left-hand side. These production instantiations are recorded in a *conflict set of production instantiations.*

The inference engine performs the so-called *recognize-act cycle*, consisting of three separate phases:

1. *The match phase.*

   The left-hand sides of the productions in the production memory are matched against the elements in the working memory, yielding a conflict set of production instantiations, as described above.

2. *The selection phase.*

   From the conflict set, one production instantiation is selected for execution. This selection is based on the time-tags of the matching working memory elements recorded in the production instantiations. In general, the process of selecting a suitable production instantiation from the conflict set may be based on several different criteria and is called *conflict resolution.* When conflict resolution does not yield an instantiation, the inference is halted.

3. *The action phase.*

   The actions specified in the right-hand side of the production of the instantiation selected from the conflict set are executed.

The three phases constitute one full inference cycle, which is iterated over and over again.

The Rete Match Algorithm

The production system paradigm has proven to be extremely powerful in practice. For larger applications, however, efficiency is a major problem: especially the match phase of the recognize-act cycle tends to become the major consumer of computing resources, even so to an unacceptable extent. To overcome this problem, efficient match algorithms have been developed, such as the *Rete match algorithm*, [Forgy, 1982], and the *Treat algorithm*, [Miranker, 1987]. The basic idea of these algorithms is to save and store match information for reuse, between successive iterations of the recognize-act cycle of the inference engine. Reusing this information allows for the match phase of a new inference cycle to be directed to the *changes* in the working memory resulting from the previous cycle, thus saving on unnecessary re-matching. We briefly review the Rete match algorithm.

To support the match phase of the recognize-act cycle, the productions from the production memory of a bottom-up inference system are compiled into a discrimination network, called the *Rete network.* Such a network basically consists of four types of node:

- the network contains a unique *root node*;
- for each condition element of a production, the network contains a sequence of *one-input nodes* representing the features the inference engine has to test when trying to match the condition element with a given working memory element — the one-input nodes represent the *intra-element* features a matching working memory element is required to exhibit;
- for a production with n condition elements, $n \geq 2$, the network contains $n-1$ *two-input nodes* representing the features the inference engine has to test when trying to match a set of interrelated condition elements with a set of working memory elements — the two-input nodes represent *inter-element* features of these working memory elements;
- for each production, the network contains one *terminal node* representing the entire right-hand side of the production.

Note that, since the match phase of the recognize-act cycle is concerned with the left-hand sides of productions only, there is no need to split up the right-hand sides of the productions into actions and represent these separately.

In addition to the four types of node mentioned above, two types of special node may occur in a Rete network. The first type of special node is used to represent tests to be performed in the case that a variable occurs more than once in the same condition element; such a node specifies an intra-element feature and therefore is a special one-input node. The second type of special node is used to represent negated condition elements. A negated condition element is represented in basically the same manner as a positive condition element is represented: the intra-element features specified in the condition element are represented in a sequence of one-input nodes — its negation, however, is represented in a special two-input node. We will not elaborate any further on these types of node.

A Rete network is used for storing match information. To this end, the network is enhanced with several memory locations:

- with the outgoing arc of the last node of a sequence of one-input nodes is associated an *α-memory* — an α-memory stores all working memory elements matching the condition element represented by the preceding sequence of one-input nodes;
- with the outgoing arc of each two-input node is associated a *β-memory* — a β-memory stores all combinations of working memory elements matching the set of condition elements encompassed by the two-input node.

Note that the last β-memory in the representation of a production stores the instantiations of this production given the current contents of the working memory.

The inference engine exploits the Rete network built from the productions from the production memory not only for storing match information, but also as a means for efficiently computing changes to the working memory. To this end, it views the Rete network as a kind of *data flow* network. Recall that after selecting a production instantiation from the conflict set, the inference engine executes the actions specified in the right-hand side of the production involved. For every action inducing a change to the current working memory, the inference engine creates one or more *tokens* of the following form:

$$(\langle label \rangle\ \langle wme \rangle)$$

where *label* indicates the type of action to be performed on the working memory element *wme*. For the execution of a `make`-action, the label `+` is used to indicate that *wme* has to be added to the working memory; for a `remove`-action, the label `-` is used to indicate that *wme* has to be removed from working memory. A `modify`-action is treated by the inference engine as being composed of a `remove`- and successive `make`-action; consequently, the inference engine creates two tokens for the execution of such an action. The inference engine propagates the tokens created throughout the network in the match phase of the recognize-act cycle, performing tests as it goes along and effectuating the changes prescribed by the tokens in the α- and β-memories encountered.

Henceforth, we will speak of a 'production system' meaning a rule-based bottom-up inference system written in OPS5.

2.2 Reason Maintenance Systems

Justification-based reason maintenance systems are devised to support problem solvers that perform some type of non-monotonic reasoning, by keeping track of their beliefs, [Doyle, 1979; Lukaszewics, 1990]. The basic idea is that the reason maintenance system keeps a historical

record of the data the associated problem solver has been working on and the inferences it has made; to this end, the problem solver informs the reason maintenance system of its activities. The reason maintenance system in turn informs the problem solver about its current beliefs. The two co-operating systems are strictly separated and do not have any knowledge as to each other's internal working; the only communication between the two systems is about inferences and beliefs. Deciding on which information to pass on to and fro is known as the *encoding problem.*

The Dependency Network

A reason maintenance system records the information it receives from its associated problem solver in a so-called *dependency network*, [Elkan, 1990]. Such a dependency network consists of *nodes*, representing problem solver data, and *justifications*, representing inferences made by the problem solver. Each node in the network is assigned a *belief status* indicating whether or not the data it represents is currently believed by the problem solver; these belief statuses are denoted as *in*, representing belief, and *out*, representing lack of belief, respectively.

A justification j is an expression of the following form:

$$j = \langle \{p_1, \ldots, p_m\} | \{q_1, \ldots, q_n\} \rightarrow r \rangle$$

where $m \geq 0$, $n \geq 0$; $p_1, \ldots, p_m, q_1, \ldots, q_n$ and r denote nodes in the dependency network. The nodes $p_1, \ldots, p_m$ are called the *monotonic supporters* of the justification j; the nodes $q_1, \ldots, q_n$ are called the *non-monotonic supporters* of j, and r is called its *consequence*. A pair (p_i, r), $i = 1, \ldots, m$, is called a *monotonic edge* in the network; a pair (q_j, r), $j = 1, \ldots, n$, is called a *non-monotonic edge*. A justification without any non-monotonic supporters, that is, where $n = 0$, is called *monotonic*; a justification including one or more non-monotonic supporters, that is, where $n \geq 1$, is called a *non-monotonic* justification. To conclude, we discern a special type of justification: a justification without any supporters is called a *premise.*

The justification j shown above expresses that belief in $p_1, \ldots, p_m$ and lack of belief in $q_1, \ldots, q_n$ form a valid reason for believing r. More formally, a justification is said to be *valid* if each of its monotonic supporters is assigned the belief status *in*, and each of its non-monotonic supporters is assigned the status *out*. By definition, a premise is valid.

Dependency networks are often represented graphically; the basic building blocks for depicting dependency networks are shown in Figure 1.

Reason Maintenance

The justifications in a dependency network are used by the reason maintenance system for computing the belief statuses of the nodes. The basic idea is that a node is assigned the belief status *in* if there is at least one valid justification having the node for its consequence; otherwise, the node is assigned the belief status *out*. In assigning belief statuses, the reason maintenance system takes care not to assign the belief status *in* to a node whose valid justifications depend monotonically on this precise status, that is, the reason maintenance system does not admit monotonic circular reasoning. An assignment of belief statuses to all nodes in the network is called a *labeling* of the network; the belief status assigned to a node is called its *label.*

As a consequence of the inferences reported by the problem solver to the reason maintenance system, the dependency network changes dynamically. The changes to the network

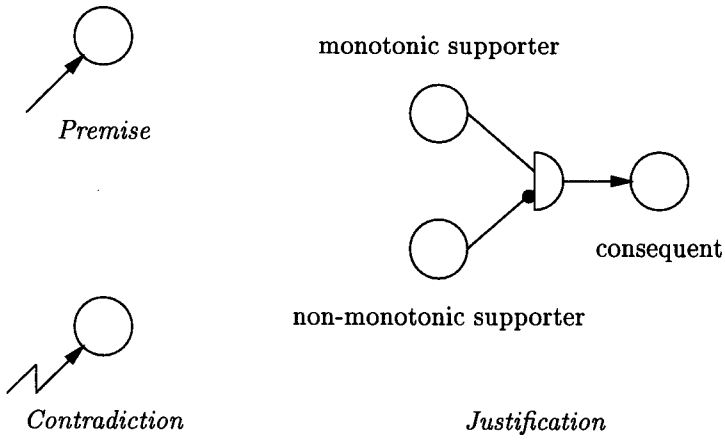

Figure 1: Building Blocks for a Dependency Network.

may cause the belief statuses of several nodes in the network to become inaccurate. To update the belief statuses of these nodes, the reason maintenance system applies a so-called *reason maintenance procedure.* The most well-known reason maintenance procedure is *Goodwin's labeling procedure.* This procedure assigns appropriate labels to the nodes of a given dependency network in a recursive fashion; the procedure comprises three separate phases that are repeated alternately until all nodes of the network have been assigned a label. Here we briefly review Goodwin's labeling procedure; for further details, the reader is referred to [Goodwin, 1982].

In the first phase of Goodwin's labeling procedure, a label is assigned to each node whose label is determined uniquely by the labels that have previously been assigned to other nodes in the network. So, a node that already has a valid justification is assigned the label *in*, and a node that has invalidated justifications only or no justifications at all is assigned the label *out.* This phase is repeated recursively until no more nodes can be assigned a label. Execution of this first phase of the procedure will not always yield a complete labeling of a dependency network. If this phase leaves several nodes unlabeled, then the network contains a loop in which the labels of the nodes are dependent upon themselves.

In the second phase of the procedure, all yet unlabeled nodes that can never be assigned an *in* label are determined; these nodes are assigned the label *out.* This phase takes care of labeling the yet unlabeled nodes that occur in a monotonic loop, that is, a loop composed of monotonic edges only. The first and second phase are repeated alternately until no more new labels can be assigned. If there are still some nodes left without a label, then the network contains a non-monotonic loop, that is, a loop comprising at least one non-monotonic edge.

In the third phase, which is known as the *Swedish Colouring procedure*, the labeling of yet unlabeled nodes appearing in a non-monotonic loop is taken care of. The procedure arbitrarily selects one such node and assigns it a label, either *in* or *out.* Each monotonic supporter of this node appearing in the loop is assigned the same label as the node itself; its non-monotonic supporters are assigned the opposite label. In this backwards fashion, the remaining nodes of the non-monotonic loop are labeled until the node first selected is reached again. If the loop comprises an *even* number of non-monotonic edges, then the nodes in the loop have been labeled successfully. However, if the loop comprises an *odd* number of non-monotonic edges, then pursuing the Swedish Colouring procedure would result in two different labels being assigned to the same node: the procedure is not able to find a correct labeling for the

nodes of the loop and terminates with an *error message.*

If Goodwin's labeling procedure terminates with a labeling, then this labeling is correct; however, if the procedure terminates with an odd loop error message, then it is not guaranteed that there does not exist a correct labeling for the network. It is noted that the problem of finding a correct labeling for a dependency network has been proven to be NP-hard, [Elkan, 1990]. To conclude, it is mentioned that the labeling procedure generally is applied in an incremental fashion. In response to a change in the dependency network, the nodes whose labels are affected by the change are identified; these nodes are unlabeled and subsequently relabeled using Goodwin's labeling procedure.

Consistency Maintenance

Besides maintaining the validity of the information represented in its dependency network, a reason maintenance system also ensures the consistency of this information. A dependency network may contain a special type of node, called a *contradiction node*, universally denoted as $\perp$ in this paper. Such a contradiction node is of special interest to the reason maintenance system. Contradiction nodes are taken to represent contradictory information that should never be believed by the associated problem solver; the reason maintenance system therefore enforces these nodes to be assigned the label *out.*

The reason maintenance procedure employed by the system does not pay any special attention to the contradiction nodes in a dependency network, and therefore may yield a labeling in which a contradiction node is assigned the label *in*; such a labeling is called *inconsistent.* When an inconsistent labeling is found during maintenance, the reason maintenance system subsequently applies a so-called *consistency maintenance procedure.* Such a procedure takes a dependency network and its (inconsistent) labeling, and (slightly) modifies the network in such a way that the reason maintenance procedure will yield a consistent labeling for the new network that reflects as much of the old labeling as possible. The most well-known consistency maintenance procedure is *dependency directed backtracking.* Here, we briefly review this procedure; for further information, the reader is referred to [Doyle, 1979].

Dependency directed backtracking is motivated by the observation that any valid non-monotonic justification may be invalidated by adding a valid justification for one of its non-monotonic supporters. The basic idea of the procedure is as follows. For a contradiction node $\perp$ having the label *in*, the procedure selects a node r having the following properties:

- node r is assigned the label *in* in the current (inconsistent) labeling of the network;
- there is a valid *non-monotonic* justification for node r in the network;
- node r contributes to the label *in* of the contradiction node through a trail of *monotonic* edges only;
- node r is *maximal* in the sense that there is no other node on this trail of monotonic edges having the above properties.

The selected node r is called the *culprit* for the contradiction $\perp$. Subsequently, one of the non-monotonic supporters of a valid non-monotonic justification for the culprit r is selected; this node is called the *denial* of r. The dependency directed backtracking procedure concludes by creating a suitable valid justification for the selected denial — in the following, we will refer to the justification created as the *backtracking justification*; for details of the precise justification

created, we refer once more to [Doyle, 1979]. As a result of the addition of the backtracking justification to the dependency network, the denial will be assigned the label *in* and thus invalidate the selected non-monotonic justification for r. In the case that this justification was the only valid one for node r, r will now be assigned the label *out*. However, node r may have more than one valid justification. Dependency directed backtracking therefore may have to be repeated as long as the label of $\perp$ has not changed from *in* to *out*. Note that in this procedure, the culprit as well as the denial are selected arbitrarily.

Henceforth, we will speak of a 'reason maintenance system' meaning a justification-based reason maintenance system as described above.

3 The Encoding

The basic problem in designing an architecture in which a given problem solver and a supporting reason maintenance system are to co-operate is the *encoding problem*. This problem consists of two subproblems. The first subproblem is to decide on which problem solver information is to be sent to the reason maintenance system and how it is to be presented; this part of the encoding problem for our architecture is the topic of Section 3.1. The second subproblem is to decide on which reason maintenance information is to be sent to the problem solver and how that information is to be presented. In Section 3.2 we address this problem for our architecture. The question as to how the reason maintenance system and the problem solver interact is postponed until Section 4.

3.1 The Encoding of Production System Information

To address the encoding of information from the production system to be sent to the reason maintenance system, we recall that one of the tasks of the reason maintenance system is to keep a historical record of the data the production system has been working on. We observe that all data ever relevant to the reason maintenance system are included in the working memory maintained by the production system, or have once been present there during problem solving. The basic idea now is to take a node in the dependency network of the reason maintenance system to represent a working memory element; its belief status is taken to represent presence or absence from working memory. To be more precise,

- a node in the dependency network is labeled *in* if and only if its associated element is present in the working memory maintained by the production system;
- a node is labeled *out* if and only if its associated element is absent from working memory.

Note that at all times each element in the current contents of the working memory has associated a corresponding node in the dependency network. However, not every node in the dependency network has a counterpart in the current working memory.

The reason maintenance system not only records the production system's *data* but also its *inferences*. It will be evident that for the purposes of reason and consistency maintenance every inference made by the production system that induces some change to the contents of the working memory has to be encoded and passed on to the reason maintenance system. In Section 3.1.1 we discuss how inferences from the production system involving `make`-actions only are encoded; Section 3.1.2 addresses the encoding of `remove`-actions. We have mentioned before that the inference engine of the production system treats a `modify`-action as being

composed of a `remove`- and successive `make`-action; `modify`-actions will therefore not be dealt with explicitly.

3.1.1 The Encoding of `make`-actions

Inferences of a production system arise from the execution of (instantiations of) productions. In this section, we focus on the encoding of inferences arising from productions involving `make`-actions only. In doing so, we distinguish between several types of production:

- productions specifying positive condition elements only;
- productions involving at least one negated condition element;
- top-level `make`-instructions.

These types of production will be dealt with separately.

Productions Specifying Positive Condition Elements Only

We begin by considering a production of the following form:

$$\texttt{(p a}_1 \cdots \texttt{a}_m \texttt{ --> (make b}_1\texttt{)} \cdots \texttt{(make b}_n\texttt{))}$$

where $\mathtt{a}_j$, $j = 1, \ldots, m$, $m \geq 1$, denotes a positive condition element, and $\mathtt{b}_i$, $i = 1, \ldots, n$, $n \geq 1$, denotes a new working memory element. Now assume that an instantiation of this production is executed by the inference engine of the production system.

The execution of the `make`-actions specified in the right-hand side of the production induces the insertion of n new elements into the current working memory. We recall that the production system assigns each element that is added to the working memory a unique time-tag; the new elements $\mathtt{b}_i$ (including their respective time-tags) inserted due to the execution of the production shown above therefore cannot have been present in the working memory earlier in the problem solving process. Hence, the dependency network of the reason maintenance system does not contain any counterparts for these elements as yet. The reason maintenance system creates new nodes b_i corresponding to these elements and inserts them into its dependency network.

Belief in the validity of the new elements added to the working memory arises from the execution of the corresponding production instantiation. This inference therefore has to be encoded into justifications for the nodes b_i corresponding to these elements. Exploiting the observation that the inference has been made possible by the presence of working memory elements matching the condition elements $\mathtt{a}_j$, $j = 1, \ldots, m$, of the production, we propose the following encoding. For each node b_i, a justification of the following form is created:

$$\langle \{a_1, \ldots, a_m\} | \varnothing \rightarrow b_i \rangle$$

where a_j is the node in the dependency network that corresponds with the working memory element matching the condition element $\mathtt{a}_j$. Note that for the production shown above a set of n justifications is created each specifying the same monotonic supporters and an empty set of non-monotonic supporters. Also observe that each node in the dependency network corresponding with a working memory element can have *at most one* such justification.

Productions Involving At Least One Negated Condition Element

Attention will now be focused on the encoding of inferences arising from the execution of productions with negated condition elements. Consider a production of the following form:

$$\texttt{(p a}_1 \cdots \texttt{a}_{k-1} -\texttt{a}_k \texttt{ a}_{k+1} \cdots \texttt{a}_m \texttt{ --> (make b}_1\texttt{) ... (make b}_n\texttt{))}$$

where $m \geq 2$, $n \geq 1$, and where the k-th condition element is the only negated one, $2 \leq k \leq m$. Now assume that an instantiation of this production is executed by the inference engine of the production system. The execution of the **make**-actions specified in the right-hand side of this production once more induces the insertion of n new elements into the working memory. As before, the reason maintenance system creates new nodes b_i corresponding to these elements and inserts them into the dependency network.

The belief in the validity of the new elements is based not only on the presence of some specific information in the working memory, but also on the *absence* of some other information. From the production having been selected and executed, we know that its negated condition element has succeeded, and therefore that matching working memory elements are lacking from the current working memory. In order to represent this feature, the reason maintenance system introduces a special node $\hat{a}_k$ corresponding to the negated condition element into its dependency network; initially, this node has no justification and therefore is assigned the label *out*. The inference made by the problem solver is then encoded as follows. For each node b_i, a justification of the following form is created:

$$\langle \{a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_m\} | \{\hat{a}_k\} \rightarrow b_i \rangle$$

where a_j once more is the node in the dependency network that corresponds with the working memory element matching the condition element $\texttt{a}_j$, $j = 1, \ldots, k-1, k+1, \ldots, m$.

The node $\hat{a}_k$ introduced to correspond to the negated condition element is special in the sense that, unlike other nodes in the dependency network, it does not represent a single working memory element: the node may be looked upon as a generic description of a *collection* of elements none of which is allowed to occur in the working memory. Note that this description may depend on the condition elements $\texttt{a}_1, \ldots, \texttt{a}_{k-1}$ preceding the negated condition element in the production's left-hand side due to the occurrence of variables.

Initially the justification created to correspond to the inference described above is valid. Now suppose that a new working memory element $\texttt{a}_k$ that 'matches' the negated condition element is inserted into the working memory. As before, the reason maintenance system creates a new node a_k and includes it with the proper justification in its dependency network. Since the addition of this new working memory element causes the left-hand side of the production shown above to fail, the justification representing the inference from this production should be invalidated. This is achieved by adding a valid justification for the node $\hat{a}_k$ to the dependency network:

$$\langle \{a_k\} | \varnothing \rightarrow \hat{a}_k \rangle$$

Note that, as $\hat{a}_k$ models a collection of working memory elements, it can have more than one such justification.

Top-Level make-Instructions

To conclude, we consider encoding the execution of a top-level **make**-instruction. Consider the following instruction:

```
(make b)
```

The execution of this **make**-instruction enforces the insertion of a new element **b** into the production system's working memory. Just like before, the reason maintenance system inserts a new node b corresponding to this new working memory element into its dependency network. Since the new working memory element results from the execution of a top-level instruction, the production system has not attached any prerequisites to its belief in the validity of this element. This unconditional belief is represented by encoding the counterpart of the element, that is, the node b, as a premise in the dependency network of the reason maintenance system. We will return to this encoding presently.

3.1.2 The Encoding of remove-actions

In the previous section, we have discussed how inferences of the production system involving **make**-actions only may be encoded; the encoding proposed is rather straightforward. In this section, we will see that the encoding of inferences involving **remove**-actions is much more complex. Before turning to this encoding, we observe that when a rule-based bottom-up inference system is supported by a reason maintenance system the use of explicit **remove**-actions for correct program behaviour may no longer be necessary: the reason maintenance system will take care of the removal of invalidated working memory elements by employing its reason and consistency maintenance procedures.

In discussing the encoding of inferences arising from the execution of productions that induce the removal of working memory elements, we distinguish between two types of production:

- productions specifying at least one **remove**-action;
- top-level **remove**-instructions.

These types of production will be dealt with separately.

Productions Specifying At Least One remove-Action

We begin by considering a production of the following form:

```
(p a1 ··· am --> (remove b))
```

where $\mathtt{a}_i$, $i = 1, \ldots, m$, $m \geq 1$, denotes a positive condition element, and **b** denotes an element that is present in the current working memory. Now assume that an instantiation of this production is executed by the inference engine of the production system.

The inference made by the production system may be looked upon as specifying an explicit reason for *disbelief* in the validity of the working memory element that is to be removed. Since this element is present in the current contents of the working memory, the dependency network of the reason maintenance system includes a corresponding node. Taking the basic idea of our encoding, removal of the element from working memory should result in its counterpart being assigned the label *out*. However, a justification-based reason maintenance system does not provide explicit means for representing reasons for disbelief: it is possible to enforce a node's label changing from *in* to *out* only through laborious encoding.

For the purpose of representing a reason for disbelief we introduce the notion of a *negative justification*. For the production shown above, the negative justification takes the following form:

$$\langle \{a_1, \ldots, a_m\} | \varnothing \nrightarrow b \rangle$$

This justification expresses that belief in $a_1, \ldots, a_m$ forms a valid reason for *dis*believing b. The negative justification is presented to the reason maintenance system as a sequence of ordinary justifications:

$$\langle \{a_1, \ldots, a_m\} | \varnothing \rightarrow \neg b \rangle$$

$$\langle \varnothing | \{b\} \rightarrow \neg b \rangle$$

$$\langle \varnothing | \{\neg b\} \rightarrow b \rangle$$

$$\langle \{b, \neg b\} | \varnothing \rightarrow \bot \rangle$$

where $\bot$ is a contradiction node. The graphical representation of this set of justifications is shown in Figure 2.

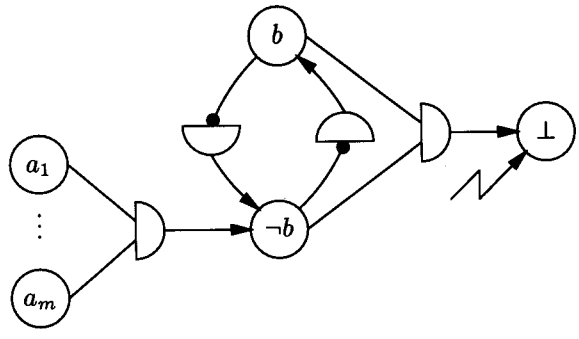

Figure 2: The Representation of the Negative Justification $\langle \{a_1, \ldots, a_m\} | \varnothing \nrightarrow b \rangle$.

The node $\neg b$ is a new node to be inserted into the dependency network by the reason maintenance system: it represents the logical negation of node b in the sense that one and only one of b and $\neg b$ can be assigned the label *in*. This property is expressed in the last three justifications shown above. The second and third justification ensure that at least one of the nodes b and $\neg b$ is assigned the label *in*. Now observe that both b and $\neg b$ may have been assigned the label *in*, because b's label may be based on some previously added, valid justification; in fact, this situation arises when the justifications modelling the execution of the `remove`-action are added to the network. This situation is noticed to be inconsistent by means of the last justification. Upon encountering this inconsistency, the reason maintenance system will perform consistency maintenance. We assume here that the consistency maintenance procedure used enforces the correct labeling of the nodes b and $\neg b$; we return to consistency maintenance in Section 4.

Until now we have looked at productions that specify positive condition elements only and include just one `remove`-action. The encoding of an inference arising from a production involving negated condition elements is the same as discussed in Section 3.1.1. In addition, we observe that as the different actions specified in the right-hand side of a production are dealt with separately in the encoding, multiple `remove`-actions and the inclusion of additional `make`-actions in the right-hand side of a production do not pose any additional problems.

Top-Level remove-Instructions

Before turning to the encoding of the execution of top-level **remove**-instructions, we observe that the presence of **remove**-actions has impact on the encoding we have proposed in Section 3.1.1 for the execution of top-level **make**-instructions. Recall that we have encoded a new element that is inserted into the working memory as a result of the execution of a top-level **make**-instruction as a premise. A premise by definition can never be disbelieved: its label will never change from *in* to *out*. From the duality in meaning between the elements of the working memory and the nodes of the dependency network, we have that the counterpart of the premise in working memory can never be removed by the problem solver. However, the OPS5-language allows for removing elements from working memory that have been inserted by means of a top-level **make**-instruction. It will be evident that in the presence of **remove**-actions the execution of a top-level **make**-instruction should not be encoded as a premise.

The basic idea is to encode a new element that is inserted into the working memory by means of a top-level **make**-instruction as a kind of *assumption* that is believed as long as there is no reason for disbelieving it. The inference arising from the execution of a top-level **make**-instruction

```
(make b)
```

is now encoded as

$$\langle \varnothing | \{\bar{b}\} \rightarrow b \rangle$$

where $\bar{b}$ is a newly added dummy node related to b. Initially, node $\bar{b}$ has no justifications and therefore is assigned the label *out*. As a result, node b is assigned the label *in*.

Now suppose that the element **b** just added to the working memory is removed further on in the problem solving process. This removal is encoded in the way described above, resulting in a contradiction node having the label *in* in the dependency network. The consistency maintenance procedure is called upon to resolve the contradiction. We assume here that the consistency maintenance procedure designates node b as the culprit for the contradiction and node $\bar{b}$ as its denial; we will return to this observation shortly. The consistency maintenance procedure inserts a new, valid justification for $\bar{b}$ into the network, thus invalidating the justification for b modelling the execution of the top-level **make**-instruction. As this was the only valid justification for node b in the dependency network, b will be assigned the label *out*.

The encoding of the execution of a top-level **remove**-instruction is now rather straightforward. The inference arising from

```
(remove b)
```

is simply encoded as a negative premise for node b in the dependency network, thus forcing the reason maintenance system to assign it the label *out*. Note that, as a consequence of the time-tag mechanism employed by the production system, there can never arise any need to change this node's label to *in* again.

3.2 The Encoding of Reason Maintenance Information

In the previous section we have addressed the problem of deciding on which information from the production system is to be passed on to the reason maintenance system and how it is to be encoded. Once encoded the information is entered into the dependency network of the

reason maintenance system and may result in changes to the labeling of some of the nodes in the network. As these reflect changes in the beliefs of the production system, the production system has to be informed of them to allow for the updating of its information base.

To address this part of the encoding problem, we recall that for incorporating changes into its working memory the inference engine of the production system makes use of tokens; these tokens are created in the act phase of the recognize-act cycle and are processed in the subsequent match phase, where all necessary changes to the α- and β-memories of the Rete network are effectuated. The basic idea of our encoding is to create a token for each change to the labeling of the nodes in the dependency network of the reason maintenance system that is of relevance to the production system; a change in the belief status of a node that was added solely for the internal working of the reason maintenance system is not relevant to the production system and therefore is not encoded. For all relevant changes to the labeling of the nodes in the dependency network, the following encoding is used:

- a token (+ *datum*(n)) is created for either a newly added relevant node n labeled *in*, or an existing relevant node n that had its label changed from *out* to *in* as a result of maintenance;
- a token (- *datum*(n)) is created for an existing relevant node n that had its label changed from *in* to *out*, either due to an explicit **remove**-action or as a result of reason maintenance;
- no token is created for a newly added node n labeled *out*.

The tokens created are processed in the match phase of the recognize-act cycle, thus forcing the production system to explicitly take care of the validity of its information base. We will further elaborate on this observation in the next section.

4 The Architecture of Interaction

A problem solver and a supporting reason maintenance system co-operate in a highly modular architecture in which the two systems are strictly separated and do not have any knowledge of each other's internal working. The two systems interact through an interface taking care of the proper encoding of the information passed to and fro. In general, however, the separation of the two systems cannot be as strict as is often suggested: for example, the problem solver has, at least to some extent, to synchronize its problem solving procedures with the maintenance procedures of its associated reason maintenance system. In addition we observe that in many applications a more efficient behaviour of the overall system may be achieved if the problem solver and its reason maintenance system are to some degree fine-tuned to each other's internal working. We will elaborate on these observations in view of our architecture in Section 4.1 and 4.2; in Section 4.3, we briefly touch upon some implementational issues.

4.1 The Enhanced Recognize-Act Cycle

The duality in meaning between the elements of the working memory of a rule-based bottom-up inference system and the nodes of the dependency network of a supporting reason maintenance system introduces the need of careful *synchronization* of operations on data. More in specific, care has to be taken that maintenance has been performed and effectuated before

the production system inspects the contents of its working memory in the match phase of the recognize-act cycle. To achieve the required synchronization, the recognize-act cycle is enhanced by a fourth phase, called the *maintenance phase.* This maintenance phase is executed immediately after the act phase has been completed and just before the match phase of the next inference cycle is entered. The *enhanced recognize-act cycle* now consists of the following phases:

1. *The match phase.*

   The tokens yielded by the maintenance phase of the previous iteration of the enhanced cycle are processed, resulting in an updated conflict set of production instantiations.

2. *The select phase.*

   From the conflict set, one production instantiation is selected for execution; in the case that no instantiation is selected, the inference halts.

3. *The act phase.*

   The actions specified in the right-hand side of the production of the instantiation selected from the conflict set are executed, resulting in a set of tokens.

4. *The maintenance phase.*

   The selected production instantiation and the tokens yielded by the act phase are encoded and sent to the reason maintenance system as described in the previous section. Subsequently, the reason maintenance system is called upon to perform maintenance. The information received from the reason maintenance system is a set of tokens.

Note that in the match phase of this enhanced recognize-act cycle not only the changes to the working memory induced by the execution of the selected production instantiation are processed, but also the changes necessary for maintaining the validity and consistency of the information base.

4.2 The Reason and Consistency Maintenance Procedures

Reason maintenance systems are devised to support any type of problem solver. The reason and consistency maintenance procedures employed therefore are domain-independent and designed to be generally applicable. In fact, the modular architecture in which a problem solver and a supporting reason maintenance system co-operate derives to a large extent from the generality of the procedures of the reason maintenance system. It often is possible, however, to fine-tune the working of the reason maintenance system to specific features of the problem solver without loss of the advantages of the modularity of this architecture. In this section, we indicate how the reason and consistency maintenance procedures of the reason maintenance system may be adapted to support rule-based bottom-up inference systems.

4.2.1 The Reason Maintenance Procedure Revisited

As a consequence of the inferences reported by the production system, the dependency network maintained by the supporting reason maintenance system changes dynamically. To update the belief statuses of the nodes in the network, the reason maintenance system applies Goodwin's labeling procedure. This procedure begins by recursively assigning proper labels

to all nodes whose label is determined uniquely by earlier assigned labels and by the topology of the dependency network. If the network does not comprise any non-monotonic loops, this will yield a complete labeling for the network. However, if the network does comprise at least one non-monotonic loop, then the first phases of the procedure may leave some nodes unlabeled. In that case, labeling the dependency network is pursued by the Swedish Colouring procedure. This procedure selects a node from the loop and assigns it an arbitrary label. If the loop is even, the procedure will be able to assign correct labels to all nodes in the loop and the remainder of the network is labeled subsequently. If the loop is an odd non-monotonic loop, however, the procedure will terminate with an error message.

We observe that Goodwin's labeling procedure only has freedom of choice in the Swedish Colouring procedure for labeling non-monotonic loops: all label assignments in the early phases of Goodwin's labeling procedure are a direct result of the semantics of the network. For investigating fine-tuning the reason maintenance procedure to the working of a rule-based bottom-up inference system, we consider the behaviour of the Swedish Colouring procedure on non-monotonic loops. In doing so, we distinguish between two types of non-monotonic loop:

- non-monotonic loops arising from the encoding of negated condition elements;
- non-monotonic loops arising from the encoding of the execution of **remove**-actions.

We consider these types of loop in isolation first, and subsequently analyse their interaction.

Non-Monotonic Loops From Negated Condition Elements

We begin by considering the execution of an OPS5-program that is composed of productions that do not include any **remove**-actions. The dependency network maintained by the reason maintenance system may comprise non-monotonic loops. We observe that, due to the absence of **remove**-actions in the productions involved, loops can only arise from the encoding of negated condition elements. To support this observation, we note that each node representing a working memory element has one and only one justification; so, no loop involving only such nodes can exist in the network. As a consequence, the following properties hold:

- each non-monotonic loop in the network involves at least one pair of nodes $\hat{a}$ and a, where $\hat{a}$ represents a negated condition element and a represents a working memory element 'matching' this condition element;
- each non-monotonic edge in a non-monotonic loop is part of the encoding of a negated condition element.

An example of such a non-monotonic loop is shown in Figure 3. This loop may have arisen as a result of the execution of instantiations of the following two productions:

```
(p -a --> (make b))
(p b --> (make a))
```

Note that in order to be syntactically correct, the first production should have included an extra condition element; for ease of exposition, however, we have omitted such an element.

We now distinguish between two different situations: the situation where the non-monotonic loop comprises an even number of non-monotonic edges, and the situation where the number

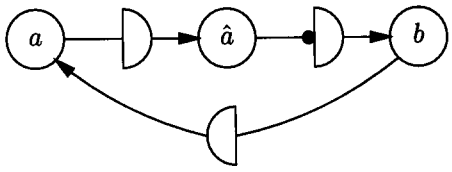

Figure 3: An Example Non-Monotonic Loop From a Negated Condition Element

of non-monotonic edges of the loop is odd. We first consider the situation where the loop comprises an *even* number of non-monotonic edges; note that Goodwin's labeling procedure will always yield a correct labeling for this loop. When the loop is first constructed, node $\hat{a}$ (or one of the other nodes corresponding to a negated condition element in the loop) has a valid justification that is no part of the loop; otherwise, the loop cannot have 'closed' as a result of the inferences reported by the production system. From this observation it follows that upon creation of the loop this node $\hat{a}$ and all the other nodes in the loop are labeled by the early phases of Goodwin's labeling procedure: both $\hat{a}$ and a are assigned the label *in*. Note that there are several correct labelings for the nodes in the loop which may arise during maintenance.

We now turn to the situation where the non-monotonic loop comprises an *odd* number of non-monotonic edges; for an example, we refer once more to Figure 3 and the two productions giving rise to the loop. Again, the labels of the nodes in this loop may already have been determined uniquely by labels assigned to nodes outside the loop; however, when the loop is first constructed this is not the case. Upon examination of the loop by the Swedish Colouring procedure, reason maintenance will terminate with an error message.

We take a closer look at the sequence of inferences of the production system represented by such a loop. It will be evident that this sequence models a derivation of the working memory element `a` based on the absence of `a` from working memory, or more in general, on the absence of elements of a collection of working memory elements including `a`. This derivation may be interpreted in two different ways:

- the derivation uncovers a real inconsistency; or
- the derivation models a proof *ad absurdum*: the purpose of the derivation is to establish the validity of `a` to be exploited for further reasoning.

Under the interpretation of the derivation as uncovering a real inconsistency, we conclude that termination of Goodwin's labeling procedure with an error message is appropriate because no proper labeling of the nodes of the loop at hand exists.

However, the productions involved in the derivation may have been designed on purpose by the programmer to allow for a proof ad absurdum. By the *Closed World Assumption*, the production system takes absence of an element `a` from working memory as falsity of `a`; the production system, as opposed to the reason maintenance system, does not have any means for modelling lack of information on `a`. Absence of an element from working memory due to falsity may be distinguished from absence due to lack of information by means of a derivation of `a` from `-a`. In this case, termination of Goodwin's labeling procedure with an error message is rather unfortunate.

The intended meaning of a derivation modelling a proof ad absurdum is easily encoded and inserted into the dependency network. The proof is encoded explicitly by adding an extra justification for node a to the network; this justification will be called an *ad absurdum*

justification. The ad absurdum justification models all information from outside the loop contributing to the validity of node a: each node that does not occur in the loop itself, yet is a supporter of a justification participating in the loop is taken as a supporter of the ad absurdum justification. Note that the ad absurdum justification bears close resemblance to the backtracking justification constructed by dependency directed backtracking.

For an *elementary* odd non-monotonic loop, that is, for an odd non-monotonic loop occurring *in isolation*, it is easily seen that the addition of an ad absurdum justification allows for labeling the nodes in the loop by Goodwin's labeling procedure and thus forestalls an error message. Now observe, however, that adding ad absurdum justifications to the dependency network destroys the property that a node representing a working memory element has at most one justification. As a result, examination of a *non-elementary* odd non-monotonic loop, that is, of several interacting (odd) non-monotonic loops, by Goodwin's labeling procedure may still result in an error message, regardless of the addition of ad absurdum justifications. The addition of ad absurdum justifications then does not serve its purpose and only increases the size of the dependency network thereby obscuring its meaning; it will be a matter of experimentation with real-life OPS5-programs to analyse the impact of the addition of ad absurdum justifications.

Non-Monotonic Loops From `remove`-Actions

In the case that `remove`-actions occur in the productions of an OPS5-program, the dependency network maintained by the reason maintenance system may comprise several different types of non-monotonic loop. It will be evident that non-monotonic loops arising from the encoding of negated condition elements as discussed above may occur in the network; in addition, the encoding of the execution of `remove`-actions as proposed in Section 3.1.2 gives rise to the introduction of non-monotonic loops in the network. These different types of loop may occur isolated from each other yet may also interact. In the sequel, we first focus our discussion on non-monotonic loops from `remove`-actions that do not interact with loops from negated condition elements and turn to the interaction of these types of loop later on. For ease of exposition, we assume that the dependency direct backtracking procedure does not introduce any loops into the dependency network.

We begin by considering the occurrence of a non-monotonic loop arising from the encoding of the execution of a single `remove`-action in isolation. We observe that such a loop involves a pair of nodes b and $\neg b$, where b represents a working memory element and $\neg b$ models its logical negation; we refer once more to Figure 2 in Section 3.1.2 for an example. We now distinguish between two different situations: the situation where the sequence of inferences of the production system leading to the introduction of node $\neg b$ into the dependency network *does not* involve the working memory element `b`, and the situation where this sequence of inferences *does* involve `b`.

We first consider the situation where the sequence of inferences of the production system leading to the introduction of node $\neg b$ into the dependency network does not involve the working memory element `b`. Then, the non-monotonic loop that has arisen comprises the nodes b and $\neg b$, and their two non-monotonic justifications only. We conclude that the loop is even and is properly labeled by Goodwin's labeling procedure; if necessary, dependency directed backtracking is called upon to resolve the contradiction.

We now turn to the situation where the sequence of inferences *does* involve the working memory element `b`. Then, an odd non-monotonic loop has arisen. To support this observation,

we note that in the dependency network the path from node b to node $\neg b$ representing the production system's sequence of inferences is composed of monotonic edges only. An example of such a loop is shown in Figure 4. When the loop is first constructed, node b has a valid justification that is no part of the loop. Initially, the loop is labeled in the early phases of Goodwin's labeling procedure and is never examined by the Swedish Colouring procedure; as a result the nodes b and $\neg b$ are *both* assigned the label *in*. Dependency directed backtracking is called upon to resolve the contradiction that has evolved. As will be discussed in Section 4.2.2, the modified dependency directed backtracking will, if possible, invalidate a valid justification for node b, leaving an odd non-monotonic loop whose nodes cannot be labeled by the early phases of Goodwin's labeling procedure. Upon examination of this loop by the Swedish Colouring procedure, an error message results.

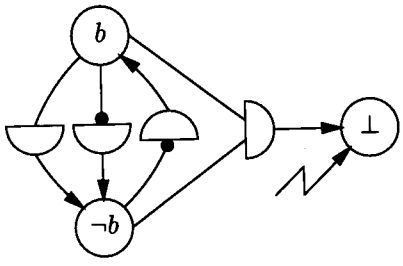

Figure 4: An Example Non-Monotonic Loop From a `remove`-Action.

Taking a closer look at the sequence of inferences of the production system represented by the loop, we once more observe that the derivation may be interpreted in two different ways:

- the derivation uncovers a real inconsistency; or
- the derivation models a proof *ad absurdum*.

Just as before, the *Closed World Assumption* underlying the production system may force us to adopt the first interpretation, in which case termination of Goodwin's labeling procedure with an error message is appropriate. However, it is likely that a proof ad absurdum has been intended by the programmer: the derivation expresses that the element `b` may not be inserted into the working memory as long as the other elements involved in the derivation are present. Termination of Goodwin's labeling procedure with an error message then is most inappropriate.

As before, the meaning of the derivation is easily encoded and inserted into the dependency network. The encoding of this second type of proof ad absurdum is quite different from the encoding we have proposed for the type of proof ad absurdum encountered previously. This second type of proof ad absurdum is modelled by directing the Swedish Colouring procedure to just assign the label *out* to node b and then to terminate. Note that by assigning the label *out* to b, one of the paths in the dependency network from b to $\neg b$ is invalidated, as this path is composed of monotonic edges and each node on the path has one justification only. Applying the early phases of Goodwin's labeling procedure now yields correct labels for the other nodes in the loop.

We note that the production system may provide two types of proof ad absurdum: the derivation of a working memory element `a` from `-a`, that is, from the absence of `a` from working memory, and the derivation of `(remove b)`, that is, of the negation of the working memory element `b`, from `b`. These derivations are similar in concept, yet are modelled quite differently

in the dependency network. This difference in modelling arises from the asymmetry of the meaning of the labels *in* and *out* employed by the reason maintenance system.

Now observe that non-monotonic loops arising from the encoding of the execution of `remove`-actions may interact. Such interacting non-monotonic loops can be handled in the same way as we have discussed for this type of non-monotonic loop in isolation; to support this observation, we note that a node $\neg b$ cannot be a supporter of any other justification than the two justifications that are part of the encoding of the `remove`-action.

We have mentioned before that besides non-monotonic loops resulting from the encoding of the execution of a `remove`-action, also non-monotonic loops resulting from the encoding of a negated condition element may occur in a dependency network. When in isolation, a non-monotonic loop arising from a negated condition element is handled in essentially the same way as discussed before. A non-monotonic loop arising from a negated condition element, however, may interact with a non-monotonic loop arising from a `remove`-action; an example of such a loop is shown in Figure 5. In the case that the negated condition element and the `remove`-action apply to the same working memory element, examination of the resulting loop by Goodwin's labeling procedure may still result in an error message regardless of the encoding proposed.

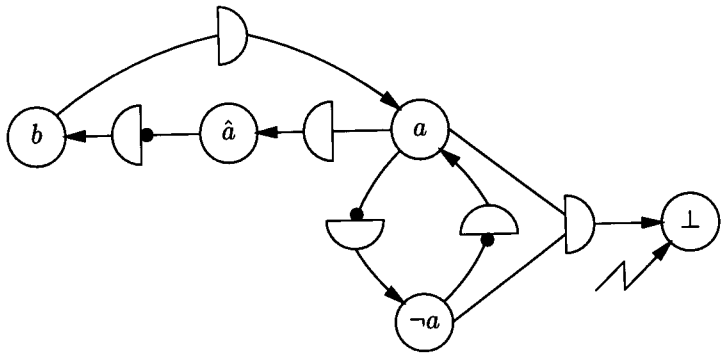

Figure 5: An Example of Interacting Non-Monotonic Loops.

4.2.2 The Consistency Maintenance Procedure Revisited

We recall that besides maintaining the validity of the information represented in its dependency network, a reason maintenance system also has to ensure the consistency of this information. To this end, the reason maintenance system applies a consistency maintenance procedure such as dependency directed backtracking: in the case that an inconsistent labeling of the dependency network is found during reason maintenance, dependency directed backtracking adds one or more backtracking justifications to the network so that the reason maintenance procedure will yield a consistent labeling for the modified network. For constructing such a backtracking justification, the consistency maintenance procedure selects a culprit for the contradiction and a denial for this culprit. The culprit as well as its denial are selected arbitrarily. In a reason maintenance system supporting a rule-based bottom-up inference system written in OPS5, however, it is possible to dictate choices for the culprit and its denial to be preferred on the basis of the encoding used.

From the encoding described in Section 3, we observe that there is only one way in which a contradiction node can have been entered into the dependency network maintained by the

reason maintenance system: only the encoding of the execution of a **remove**-action gives rise to the introduction of a contradiction node — note that the OPS5-language does not provide any explicit means of indicating a contradiction. From this encoding, we have that a contradiction node has a justification of the form

$$\langle \{b, \neg b\} | \varnothing \rightarrow \bot \rangle$$

where b is a node in the dependency network representing a working memory element, and $\neg b$ models its logical negation.

If the contradiction node has been assigned the label *in* by the reason maintenance procedure, then the justification shown above must be valid as it is the only one for this node. For this justification to be valid, the nodes b and $\neg b$ must both have been assigned the label *in*. The dependency directed backtracking procedure is called upon to resolve the contradiction that has arisen. To this end, it sets out to invalidate the justification shown above. For invalidating this justification, the dependency directed backtracking procedure may look for a culprit in the part of the dependency network contributing to the label *in* for node b or in the part giving rise to $\neg b$'s label. Suppose that the dependency directed backtracking procedure will seek the culprit for the contradiction in the part of the dependency network contributing to the label *in* for node $\neg b$. If it succeeds in finding a culprit in this part of the network, however, the inference last made by the problem solver is denied, rendering removing elements from working memory practically impossible. We conclude that the dependency directed backtracking procedure should look for a culprit in the part of the dependency network contributing to the label *in* for node b first; only in the case that no such node exists, the procedure should seek the culprit among the nodes in the part of the network preceding node $\neg b$. Note that this approach introduces a strong bias towards removing (restored) working memory elements into the dependency directed backtracking procedure.

Now suppose that the dependency directed backtracking procedure looks for a culprit in the part of the dependency network contributing to b's label as indicated above. In this part of the network, there may be several *candidate nodes* x having the following properties:

- node x is assigned the label *in* in the current (inconsistent) labeling of the network;
- there is a valid non-monotonic justification for node x in the network;
- node x contributes to the label *in* of node b through a sequence of monotonic supporters only.

A valid non-monotonic justification for such a candidate node x in the network may be one of several types:

- the justification has the form $\langle \{p_1, \ldots, p_m\} | \{\hat{p}_{m+1}, \ldots, \hat{p}_n\} \rightarrow x \rangle$, $m \geq 0$, $n \geq 1$, and represents the execution of a production (instantiation) involving at least one negated condition element and a **make**-action for the working memory element corresponding to x;
- the justification has the form $\langle \varnothing | \{\bar{x}\} \rightarrow x \rangle$ and represents the execution of a top-level **make**-instruction for the working memory element corresponding to x;
- the justification has the form $\langle \varnothing | \{\neg x\} \rightarrow x \rangle$ and is part of the encoding of the execution of a **remove**-action for the working memory element corresponding to x.

Now consider a candidate node x having a valid, non-monotonic justification of the first type mentioned above only. If this node is selected as the culprit, then its denial will have to be chosen from $\{\hat{p}_{m+1}, \ldots, \hat{p}_n\}$. Adding a backtracking justification for one of these nodes, however, will severely obscure the meaning of the dependency network: a node corresponding to a negated condition element is assigned the label *in* while no 'matching' working memory element exists. We conclude that the dependency directed backtracking procedure should better select its culprit among the set of candidate nodes having a valid non-monotonic justification of one of the last two types mentioned above, and then select the denial from this justification. Note that the culprit chosen need not be maximal in the sense of the standard dependency directed backtracking procedure.

4.3 The Sharing of Data

In the previous, it has been observed that each element in the current contents of the production system's working memory has an associated counterpart in the dependency network of the supporting reason maintenance system. The dependency network will therefore store at least the same information as is stored in the working memory of the production system. Maintaining a separate dependency network would result in redundant storage of information. To alleviate this redundancy, we propose integrating the dependency network and the working memory into one dataspace.

Before addressing the basic idea of integrating the working memory of the production system and the dependency network of the supporting reason maintenance system, we recall that the production system distributes its working memory over the memory locations of the Rete network compiled from its productions; to be more precise, the working memory elements are stored in the α-memories of the Rete network. We observe that a working memory element may occur in several different α-memories. In the sequel, we will refer to these different occurrences of a working memory element as its *α-memory occurrences*. Since a Rete network is a *static* discrimination network, one of the α-memory occurrences of a working memory element is leftmost in the network; this occurrence will play a central role in our integration and will be called the *seed occurrence* of the element.

The basic idea underlying the integration of the working memory and the dependency network is to take working memory elements as dependency nodes. There are, however, two important conceptual differences between dependency nodes and working memory elements that prohibit exploiting this idea directly:

- a node once entered into the dependency network of the reason maintenance system is never deleted from the network, whereas its associated element may be removed from the working memory maintained by the production system;
- each node in the dependency network is unique, whereas its corresponding working memory element may have several α-memory occurrences in the Rete network.

The projected integration is achieved by taking the seed occurrence of a working memory element to act as the corresponding dependency node. To this end, seed occurrences have associated a label. The labels of the seed occurrences have the same meaning as the labels of the nodes in a dependency network: the label *in* expresses the production system's belief in the working memory element, whereas the label *out* expresses lack of belief in the element. The seed occurrence will never be removed from the α-memory at hand; the Rete match algorithm simply treats an α-memory occurrence labeled *out* as non-existent.

It will be evident that the proposed integration forestalls redundant storage of information. In the worst case, however, no memory-space is saved. This worst case arises when the α-memories comprise only seed occurrences labeled *out*, that is, when all nodes in the dependency network are labeled *out* and hence the working memory does not contain any elements; such a situation is extremely rare yet not impossible. In the best case, on the other hand, the space consumption for the representation of information is halved. This best case arises when each working memory element has one α-memory occurrence only and all nodes in the dependency network are labeld *in*; this situation is also rare. For real-life applications, a saving between 0 and 50 percent in space consumption is expected for the integration proposed over the high-level encoding suggested in the previous sections; the saving actually achieved, however, is expected to be highly dependent on the application at hand.

For reasons of computational efficiency, links may be inserted into the Rete network connecting the seed occurrence of a working memory element to all α-memories containing other occurrences of this element. These links are exploited in the following sense. Suppose that a working memory element is removed from working memory, either due to problem solver activities or due to maintenance. Then, the seed occurrence of the element is assigned the label *out*; in addition, all other α-memory occurrences of the element are found by traversing the links described before and are subsequently removed. Neither the seed occurrence nor the links are removed from the Rete network. Restoring a once removed working memory element is now done simply by changing the label of its seed occurrence to *in* and restoring the other occurrences in the linked α-memories. Note that these links may also be exploited when a new element is inserted into working memory differing only in its time-tag from an element already present in working memory. We will not elaborate any further on these implementational issues; the interested reader is referred to [de Koning, 1992].

5 Implications for Bottom-up Programming

In supporting a rule-based bottom-up inference system by a reason maintenance system as described in the foregoing, our aim has been to relieve a programmer of the obligation of ensuring the validity of information contained in the working memory of the production system at all times. It will be evident that in the architecture proposed, information maintenance is taken care of by the reason maintenance system, and therefore need not be incorporated explicitly into the production system. In Section 5.1 we demonstrate the impact of support by a reason maintenance system by means of a small example production system.

In general, programming information maintenance is considered to be trivial and tedious bottom-up programming that has to be done to arrive at a usable system. Removal of the need for explicit information maintenance therefore is a major advantage of the architecture proposed to the programming style. It is expected, however, that the programming style for bottom-up inference systems will be influenced beyond information maintenance. We elaborate on this observation in Section 5.2.

5.1 An Example Production System

Supporting a rule-based bottom-up inference system by a reason maintenance system may have a major impact on the program code of such a production system. In this section, we illustrate the type of impact that may be expected by means of a small example.

Consider a judicial problem solver for murder cases. The problem solving knowledge of the system is captured in three simple rules (modified from [Lukaszewicz, 1990]):

- any person having a motive for the crime at hand who has not been proved innocent as yet, is considered a prime *suspect*;
- anyone who has an alibi for the time the crime was committed that is confirmed by a trustworthy third party, is *innocent*;
- if a person is a suspect in the murder case, then (s)he is considered *guilty*.

These rules are meant for illustrative purposes only and should not be taken too seriously.

We have developed several different OPS5-programs for this problem solver which will be discussed below. In presenting the different programs, we have omitted all declarations for reasons of brevity; the `literalize`-statements required, however, may easily be abstracted from the productions. We also like to note that the problem solver may be programmed in many different ways. Since our implementations are meant to illustrate the impact of support by a reason maintenance system, we have focused on the aspect of information maintenance only; our implementations need not necessarily be the most efficient ones.

The First Program

The first program for our judicial problem solver specifies productions for the domain-dependent rules shown above as literally as possible, that is, this program does not incorporate any code for information maintenance:

```
(p Rule1
   (has-motive ^person <x>)
  -(innocent ^person <x>)
-->
   (make suspect ^person <x>))

(p Rule2
   (has-alibi ^person <x> ^confirmed-by <y>)
   (trustworthy ^person <y>)
-->
   (make innocent ^person <x>))

(p Rule3
   (suspect ^person <x>)
-->
   (make guilty ^person <x>))
```

When this program is run with a traditional OPS5-interpreter, it may yield incorrect results; as an example, consider initialising the working memory with the following elements:

```
1: (has-alibi ^person tom ^confirmed-by john)
2: (trustworthy ^person john)
3: (has-motive ^person tom)
```

where Tom and John are two of the characters involved in the murder case at hand. After running the program, the working memory contains elements declaring Tom to be guilty as well as innocent:

```
4: (suspect ^person tom)
5: (guilty ^person tom)
6: (innocent ^person tom)
```

This observation clearly demonstrates the need for programming information maintenance. We emphasize that the above program implements our problem solver to be run with an OPS5-interpreter supported by a reason maintenance system; note that the program code is not obscured by explicit constructs for information maintenance.

To arrive at a correct, *traditional* OPS5-program for our judicial problem solver, we have used two different well-known approaches to programming information maintenance: the use of explicit `remove`-actions to delete invalidated information from the working memory and the use of *flags* to suspend and enable production evaluation.

The Second Program — The Use of Explicit `remove`-Actions

In our second program, the basic idea is to scan the working memory for invalidated elements and delete such elements as soon as they are detected making use of explicit `remove`-actions:

```
(p Rule1
   (has-motive ^person <x>)
  -(innocent ^person <x>)
-->
   (make suspect ^person <x>))

(p Rule2
   (has-alibi ^person <x> ^confirmed-by <y>)
   (trustworthy ^person <y>)
-->
   (make innocent ^person <x>))

(p Rule3
   (suspect ^person <x>)
-->
   (make guilty ^person <x>))

(p Extra1
   (innocent ^person <x>)
   {(suspect ^person <x>)<wme>}
-->
   (oremove <wme>))

(p Extra2
   (innocent ^person <x>)
   {(guilty ^person <x>)<wme>}
```

```
-->
   (oremove <wme>))
```

Now, consider running this program with a traditional OPS5-interpreter on a working memory initialised with the same working memory elements mentioned above. As soon as Tom's innocence is established by means of **Rule2**, the working memory is scanned for the presence of an element declaring Tom to be suspect by means of the production **Extra1** and for the presence of an element declaring Tom to be guilty by means of **Extra2**. When present, these elements are removed from the working memory to ensure the validity of the information contained in the memory. Note that the program may insert elements into working memory that have to be removed later on during inference. Also note that the contents of the working memory are not guaranteed to be consistent at all times throughout the problem solving process. When the above progam is compared to the first one presented, it is easily seen that information maintenance is incorporated into the program at the cost of two additional productions.

The Third Program — The Use of Flags

In the third OPS5-program for our judicial problem solver, we have used the *flag*-construct. A flag is a working memory element without any semantical meaning introduced into memory only to enforce selection and execution of a preferred set of productions — these productions typically include a condition element matching the flag. In our program, we have used a flag to suspend the execution of the production **Rule1a** until it has been established whether or not a person is innocent, that is, until **Rule2** has been considered — we like to emphasize that this approach is correct only when all information on the case at hand is available at the start of the inference.

```
(p Rule1a
   (enabled-rule ^person <x>)
   (has-motive ^person <x>)
  -(innocent ^person <x>)
-->
   (make suspect ^person <x>))

(p Rule2
   (has-alibi ^person <x> ^confirmed-by <y>)
   (trustworthy ^person <y>)
-->
   (make innocent ^person <x>))

(p Rule3
   (suspect ^person <x>)
-->
   (make guilty ^person <x>))

(p Extra1
   (character ^person <x>)
  -(has-alibi ^person <x>)
```

```
-->
   (make enabled-rule ^person <x>))

(p Extra2
   (character ^person <x>)
   (has-alibi ^person <x> ^confirmed-by <y>)
  -(trustworthy ^person <y>)
-->
   (make enabled-rule ^person <x>))
```

Consider running this program with a traditional OPS5-interpreter on a working memory initialised with the same working memory elements mentioned before. The execution of production **Rule1a** is enabled only after Tom's innocence has been evaluated by means of **Rule2**. Note that in contrast with the previous program this program does not introduce any elements (that is, other than flags) into the working memory that have to be removed later on in the inference and therefore consistency of the contents of the working memory is guaranteed throughout the problem solving process. When the above program is compared to the first one presented, it is easily seen that information maintenance is incorporated into the program at the cost of two additional productions and a modification in one of the original productions; we feel that such modifications in general tend to obscure the declarative meaning of a program and should be avoided whenever possible.

5.2 Implications for the OPS5-Programming Style

Programming styles differ from person to person. Nonetheless, several constructs are used very commonly in OPS5-programming. Many of these constructs reflect a procedural use of the OPS5-language. An example of such a construct is the use of *flags* as demonstrated in the previous section. We observe that the flag construct in general introduces an *implicit context* concept in OPS5. If the production system is supported by a reason maintenance system, then the use of flags may no longer serve the purpose of a context mechanism: as soon as a flag is removed from working memory, the inferences of the production system conditional on the flag will be invalidated by the reason maintenance system. We conclude that the general use of implicit contexts may no longer be possible. However, we feel that as the use of *implicit* contexts obscures the semantics of the productions, it should be avoided anyhow. Implicit contexts had better be made explicit; to this end, the OPS5-programming language should be enhanced with an explicit context mechanism. Note that the incorporation of an explicit context mechanism into the OPS5-programming language will in addition reduce the necessity of using **remove**-actions.

Another issue worth mentioning concerns top-level **make**-instructions. We recall from Section 3.1.2 that a working memory element that has been entered into memory by means of a top-level **make**-instruction, is treated by the reason maintenance system as a kind of assumption. As a consequence, this element may be removed from working memory as a result of maintenance. However, experience with OPS5-programming reveals that the top-level **make**-instruction is used in two different senses: it is used to enter basic facts into working memory that should never be removed and it is used to insert elements into working memory that may be removed, that is, assumptions. Note that in the former case, the top-level **make**-instruction had better be encoded as a premise. We feel that a programmer should be able

to explicitly discriminate between the different uses of the top-level **make**-instruction. To this end, the OPS5-programming language should be enhanced. A similar observation applies to **remove**-actions. In OPS5-programming practice, the **remove**-action is used either for program correctness or for 'garbage collection' for efficiency purposes. It will be evident that in the latter case the **remove**-action should not be encoded and inserted into the dependency network of the supporting reason maintenance system. To allow for 'garbage collection', a programmer has to be provided with a means of indicating the purpose of **remove**-actions. For further details on these wished-for enhancements of the OPS5-programming language, we refer to [de Koning, 1992].

We conclude by observing that only experience with programming the overall system for real-life applications will yield detailed insight into the induced programming style.

6 Conclusions

Rule-based bottom-up inference systems often employ some kind of non-monotonic reasoning to allow for dealing with incomplete information. Experience with programming this type of production system has revealed that a major part of the programming effort is spent on dynamic maintenance of the information the system is working on. One of the aims of our research has been to support a production system by a reason maintenance system to render explicit programming of this type of maintenance unnecessary. We have proposed an integrated architecture in which a production system and a supporting reason maintenance system co-operate; the encoding of the information to be exchanged between the two systems has been detailed. From a fundamental point of view, the results reported in this paper clearly reveal the feasibility of supporting a bottom-up inference system by a reason maintenance system: the encoding correctly captures the meaning of the inferences of the production system. Further research will focus on the actual implementation of our architecture. We observe that, although we departed mainly from the OPS5-programming language, most of our results apply to similar languages as well.

Reason maintenance systems are devised to support any type of problem solver; the maintenance procedures employed therefore are designed to be generally applicable. However, the reason and consistency maintenance procedures at several points provide some freedom of choice. Within this freedom of choice it is possible to fine-tune the procedures to the working of the associated problem solver without conceding generality of application. Based on this observation we have devised modified reason and consistency maintenance procedures to be incorporated in the reason maintenance system in our architecture. We feel that in most applications of reason maintenance systems it is worthwhile to examine the freedom of choice provided by the reason maintenance system and use the various options available to fine-tune the reason maintenance procedures to the problem solver at hand.

References

[Brownston *et al.*, 1985] L. Brownston, R. Farrell, E. Kant and N. Martin (1985). *Programming Expert Systems in OPS5*, Addison-Wesley, Reading, Massachusetts.

[Doyle, 1979] J. Doyle (1979). A truth maintenance system, *Artificial Intelligence*, vol. 12, pp. 232 – 272.

[Elkan, 1990] C. Elkan (1990). A rational reconstruction of nonmonotonic truth maintenance systems, *Artificial Intelligence*, vol. 43, pp. 219 – 234.

[Forgy, 1982] C.L. Forgy (1982). Rete: a fast algorithm for the many pattern / many object pattern match problem, *Artificial Intelligence*, vol. 19, pp. 17 – 37.

[Forgy, 1991] C.L. Forgy (1991). *The Design of RAL*, Production Systems Technologies, Inc.

[Goodwin, 1982] J.W. Goodwin (1982). *An Improved Algorithm for Non-Monotonic Dependency Network Update.* Research Report LiTH-MAT-R-82-23, Linköping Institute of Technology.

[de Koning, 1992] C. de Koning (1992). *Reason Maintenance in Bottom-Up Inference*, M.Sc. thesis, Utrecht University, Department of Philosophy.

[Lukaszewicz, 1990] W. Lukaszewicz (1990). *Non-Monotonic Reasoning. Formalization of Commonsense Reasoning*, Ellis Horwood Ltd., Chichester.

[Miranker, 1987] D.P. Miranker (1987). TREAT: a better match algorithm for AI production systems, *Proceedings of the Sixth National Conference on Artificial Intelligence*, pp. 42 – 47.

[Neiman & Martin, 1986] D. Neiman and J. Martin (1986). Rule-based programming in OPS83, *AI Expert.*

[Smith & Kelleher, 1988] B. Smith and G. Kelleher (1988). *Reason Maintenance Systems and Their Applications*, Ellis Horwood Ltd., Chichester.