D2.3

Reason Maintenance - State of the Art

Project title:	**Knowledge in a Wiki**
Project acronym:	KIWI
Project number:	ICT-2007.4.2-211932
Project instrument:	EU FP7 Small or Medium-scale Focused Research Projects (STREP)
Project thematic priority:	Information and Communication Technologies (ICT)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	ICT211932/LMU-Munich/D2.3/D/PU/b1
Responsible editors:	Jakub Kotowski
Authors:	François Bry and Jakub Kotowski
Reviewers:	Peter Dolog and Rolf Sint
Contributing participants:	LMU-Munich
Contributing workpackages:	WP2
Contractual delivery:	1 September 2008
Actual delivery:	1 September 2008

Abstract
This paper describes state of the art in reason maintenance with a focus on its future usage in the KiWi project. To give a bigger picture of the field, it also mentions closely related issues such as non-monotonic logic and paraconsistency. The paper is organized as follows: first, two motivating scenarios referring to semantic wikis are presented which are then used to introduce the different reason maintenance techniques.

Keyword List
reason maintenance, belief revision, reasoning, paraconsistency, explanation, rules

Project funded by the European Commission within the Seventh Framework Programme.

Reason Maintenance - State of the Art

François Bry and Jakub Kotowski

[1] Institute for Informatics, University of Munich
Email: bry@lmu.de

[2] Institute for Informatics, University of Munich
Email: Jakub.Kotowski@ifi.lmu.de

29 August 2008

Abstract
This paper describes state of the art in reason maintenance with a focus on its future usage in the KiWi project. To give a bigger picture of the field, it also mentions closely related issues such as non-monotonic logic and paraconsistency. The paper is organized as follows: first, two motivating scenarios referring to semantic wikis are presented which are then used to introduce the different reason maintenance techniques.

Keyword List
reason maintenance, belief revision, reasoning, paraconsistency, explanation, rules

Contents

1 Introduction

The Web nowadays shifts towards and provides more and more tools for online collaboration. Wikis belong to such tools and one of the goals of semantic enhancements of wikis is to further support the collaboration of users. Imagine an enterprise and a particular team working in it on a project. It is common nowadays that the team has members who meet physically only rarely or never and the tools used by different team members are diverse. In such an environment, semantically enhanced tools are even more important.

A wiki contains diverse facts and is a tool for maintaining and storing them by many different people. The facts can be used to infer additional data and metadata using reasoning. But the data upon which the inference is done are not static, they change and, in the wiki-world, versioning is expected when changes occur. In presence of reasoning and changes due to the reasoning itself and other people collaborating, the system can become complex and not transparent which is why explanation of the systems actions (and inactions) is desirable. Reason maintenance provides a basis for both explanations and revision/versioning for reasoning. As a consequence, inconsistency detection is an issue in reason maintenance. Reasoning in collaborative contexts, e.g. in wikis, calls for explanations, revisions/versioning, and detection of inconsistencies.

In the next section, a motivating scenario is presented which is followed by an overview of belief revision a reason maintenance techniques and in the final section these techniques are related to explanations and to current semantic web technologies such as RDF and OWL.

2 Motivating Scenarios

One important area was determined in social media where reasoning could be of significant value and that is tagging [9]. Lets begin with a motivating example that will outline how a user could wish to use the tagging system.

2.1 Scenario 1 - Specification review

Consider a wiki which is used to keep track of development of a software. The wiki contains various content documenting the whole development process. For example there are pages describing the requirements, use cases and feature specifications. Developers use the wiki do describe and analyze features they work on and these are later used by managers to make reviews and by documentators to produce documentation. This way they share information and cooperate. To facilitate the cooperation, the wiki could be enhanced by rule based reasoning.

Imagine a set of rules which expresses that each feature specification is a specification, each component specification is a specification, each specification that is not revised needs a review and if something that needs review has been assigned a reviewer then the thing should be on the reviewer's todo list. See the table Rules 1 which express formally the rules just described. The following scenario describes a process during which the user uses the wiki to describe a state of a software feature specification. In the diagrams, red means a newly added tag, blue means that the tag was already present. The text next to arrows indicates who takes the action indicated in the next square diagram.

Consider a page tagged by the user with a $featureSpecification$ tag. Then, thanks to rule R_1, the system infers the tag $specification$. Once this tag is present, KiWi can use rule R_3 to infer the *needsRevision* tag because the tag *revised* is not present. See Figure 1.

$$
\begin{array}{ll}
featureSpecification(x) \rightarrow specification(x) & R_1 \\
componentSpecification(x) \rightarrow specification(x) & R_2 \\
specification(x) \wedge \neg revised(x) \rightarrow needsRevision(x) & R_3 \\
needsRevision(x) \wedge reviewer(x, r) \rightarrow todo(x, r) & R_4
\end{array}
$$

Rules 1: Tagging rules 1

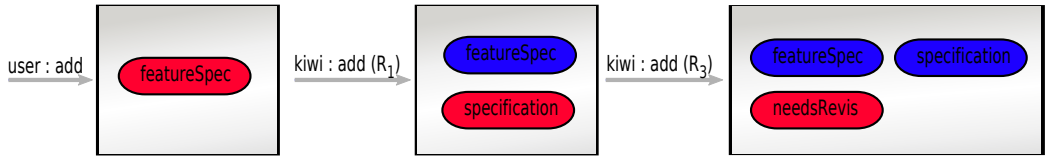

Figure 1: System infers two new tags after the user adds the "featureSpec" tag.

Then a user tags the page with the tag $reviewer(John)$, see Figure 2. Presence of this new tag enables KiWi to infer a new tag using rule R_4.

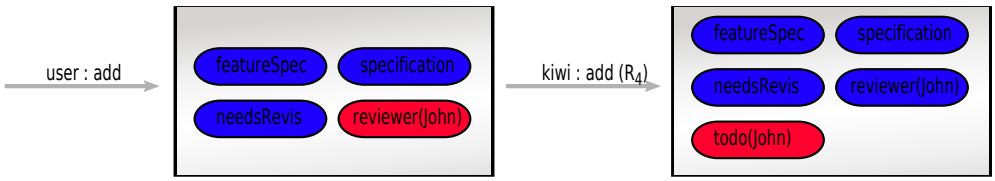

Figure 2: Deduction of tags with variables.

So far the scenario was only a straight-forward usage of tagging by user and by the system based on predefined rules. Now we will explore what interesting changes can be made in this state. There are several:

Change 1 User removes the $reviewer(John)$ tag.

Change 2 Or the user changes the $featureSpecification$ tag to $componentSpecification$ tag.

Change 3 Or the user removes the $needsRevision$ tag.

Change 4 Or the user adds a $revised$ tag.

We will explore these changes one by one, discuss them and point out some important features that KiWi will have to support in order to be able to realize this scenario.

Change 1 In software management context it can easily happen that it changes who is in charge of a task. If tagging was used to facilitate keeping track of reviewers then it would be desirable to be able to remove the *reviewer* tag and later replace it with the same tag but a different user. Then it is natural to expect that Kiwi will notice that the $todo(John)$ tag is no longer justified and will remove it, see Figure 3.

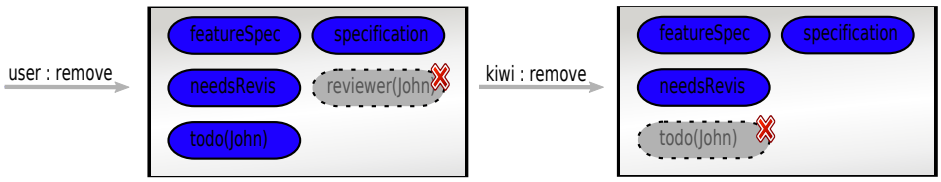

Figure 3: Removal of the "reviewer(John)" tag causes removal of the "todo(John)" tag.

Change 2 It may well happen that a user by mistake tagged something as $featureSpecification$ which in fact is a component specification. Therefore he or she changes the tag, see Figure 4.

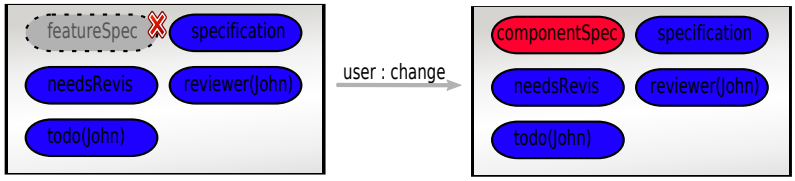

Figure 4: Change of a tag for a similar one.

Then the system has to revise all derivations based on the former $featureSpecification$ tag. This can, but doesn't necessarily have to, be done as in Change 1 by removing the first tag and adding the second one. The difference is that there clearly is space for optimization in this case. The system could be able to determine that inferences based on the $featureSpecification$ tag would actually be the same as the ones based on the $componentSpecification$ tag.

Change 3 Although there is a rule saying that each specification needs a revision, in real world it can happen that some specification is an exception to this rule. It is a question whether the system should support such a use case. If we assume that removing tags is supported in the system and if we furthermore assume that we do not want to differentiate between user generated and system generated tags then it is desirable that the user is able to remove any system generated tag as well.

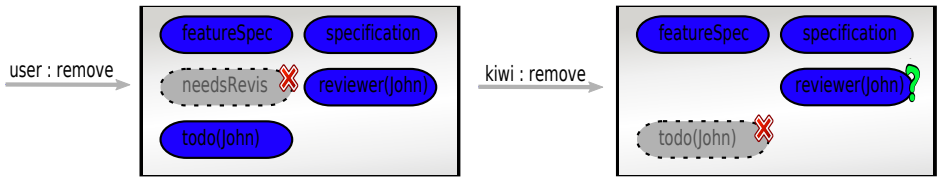

Figure 5: Removal of the "needsRevis" tag creates an exception to the rule R_3.

It is natural to expect that if the user removed the *needsRevision* tag the system would remove tags automatically generated because of the presence of the *needsRevision* tag (rather than reintroducing this tag right after the user removed it, effectively blocking removal of that tag), see Figure 5.

Change 4 The fourth possible change concludes the scenario. After the specification was reviewed, a user can tag it as revised, see Figure 6.

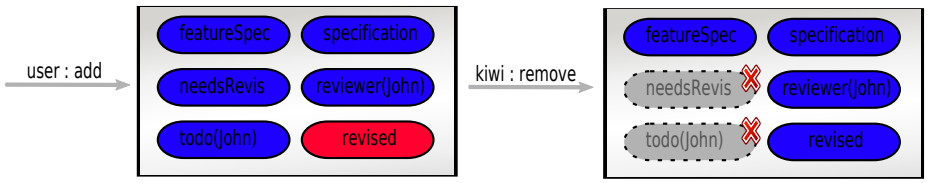

Figure 6: Non-monotonic removal of two tags after addition of the "revised" tag.

Again, the system has to notice that the $todo(John)$ tag is no longer justified because now *revised* is provable and therefore the tagging conflicts with rule R_3.

This – admittedly oversimplified – scenario indicates some requirements on reasoning and reason maintenance of the system. We see that non-monotonicity is required as the Change 4 indicates: adding a new fact removes some previously derived fact. The system has to support exceptions to rules to be able to handle Change 3. Change 1 shows the need for reason maintenance and Change 2 indicates a place for optimization in implementation of the system.

2.2 Scenario 2 - Paraconsistency

Consider another example which reveals a new requirement, see Rules 2. These rules express that a bug that is not processed is on someone's todo list. Furthermore it is inconsistent to have a bug on todo list and at the same time say that it won't be fixed or tag something both as *processed* and *notProcessed*. The tag $wontFix$ implies the tag *processed*. $wontFix$ means that the defect will not be repaired.

$$\begin{array}{ll} bug \wedge notProcessed \rightarrow todo & T_1 \\ todo \wedge wontFix \rightarrow \bot & T_2 \\ processed \wedge notProcessed \rightarrow \bot & T_3 \\ wontFix \rightarrow processed & T_4 \end{array}$$

Rules 2: Tagging rules 2

Imagine a page tagged as *bug* and *notProcessed*, see Figure 7. Because of rule T_1 kiwi will infer the *todo* tag. Then a developer decides that this bug will not be fixed and tags it as such. Then the system derives the *processed* tag using rule T_4, see Figure 8.

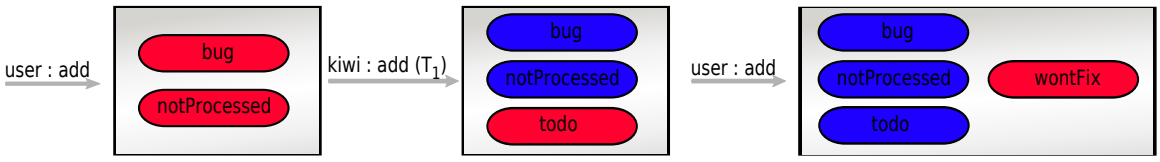

Figure 7: Process of tagging a page which describes a defective software feature.

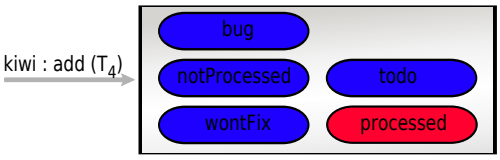

Figure 8: Resulting inconsistent state.

These actions created two inconsistencies. The first is due to the rule T_2 which says the *todo* tag is inconsistent with the *wontFix* tag, the second inconsistency originates in the presence of both *notProcessed* and *processed* tags and the rule T_3.

We can see that the obvious thing to do is to remove the *todo* and *notProcessed* tags. But how should the system know that it is better than removing the new *wontFix* tag (the user reporting that bug might well like this behaviour after all). This small example reveals a bigger problem. In collaborative environment where the system is used as a tool in a process of creative work, inconsistencies are likely to be present. Often the final decision how to resolve these inconsistencies has to be made by users. So the system should be able to work reasonably well even if inconsistencies are present and report them to user. That is, the system should be paraconsistent - inconsistency tolerant. Moreover, it is desirable that the system can pinpoint the sources of inconsistencies and track derivations based on them. See Figure 9 which depicts a derivation tree showing the inferences made by KiWi. The red and green circles mark inconsistencies and facts used to derive them. From the picture we can easily see that for example the *notProcessed* tag is source of both inconsistencies. To learn more about reasoning and inconsistencies see [9].

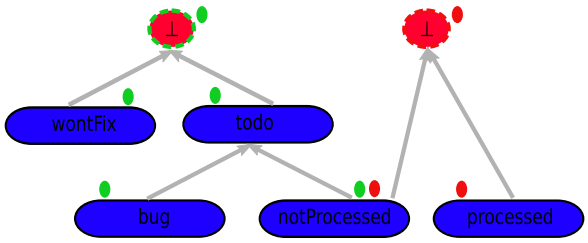

Figure 9: The internal inference tree with two inconsistencies.

3 Defeasible reasoning

The specification revision scenario in the previous section depends on the ability to defeat already derived facts. A rule supports a derivation if there is no evidence of support for its negated antecedents. In the scenario it was the rule R_3 expressing that a specification needs a revision unless it is already revised. Grigoris Antoniou, in his book about non-monotonic reasoning [4] gives another intriguing example, consider the following text: *"Smith entered the office of his boss. He was nervous."* At this point the reader of this sentence is likely to infer that Smith is the one who is nervous. But then he or she continues to read: *"After all, he didn't want to lose his best employee."* Now the reader has to revise his or her belief because

it was actually the boss who was nervous. The previous belief was based on the assumption that it is usually employees who are nervous when confronted with their boss. But in the presence of additional information, the belief turned out to be wrong and the conclusion had to be withdrawn. This kind of reasoning is called *defeasible reasoning*.

This section draws mainly from the "Defeasible reasoning" article of The Stanford Encyclopedia of Philosophy [40]. The article describes defeasible reasoning as a reasoning where the corresponding argument is rationally compelling but not deductively valid. Defeasible reasoning has been studied since the time of Aristotle and includes many subfields. Two of these subfields are belief revision and reason maintenance. This section intends to give the reader a brief high-level overview of the whole field so that it is more apparent where belief revision and reason maintenance belong.

Two main approaches can be recognized in defeasible reasoning: epistemological and logical. The epistemological approach studies defeasible reasoning as a form of inference - a process by which knowledge is increased. This approach studies the characteristics of belief change. Whereas the logical approach focuses on the consequence relation between sets of propositions. Defeasible consequence is non-monotonic (in contrast to the deductive consequence that is monotonic). Defeasible conclusion is valid if it is true in almost all of the models that verify the premises.

3.1 Epistemological approach.

The Stanford Encyclopedia of Philosophy lists three different kinds of the epistemological approach: John L. Pollock's theory of defeasible reasoning, theory of semantic inheritance networks and the AGM theory (named after its originators Alchourrón, Gärdenfors and Makinson). John Pollock's approach is constructive and effectively computable. It is based on non-deductive inferences - he defines what it is to be a *prima facie* reason for believing a proposition and what it means for a belief to be a defeater of another belief. These notions are inspired by the way a reasonable person would probably think in a situation. Semantic inheritance networks are similar to John Pollock's theory - both represent cognitive states by means of directed graphs - but are less general. In semantic inheritance networks all initial nodes represent individuals and non-initial nodes represent kinds, categories or properties and link between the nodes represents either belonging of a individual to a category or relation of "subcategory". Whereas in John Pollock's theory, the nodes represent propositions and links between them represent relation of "being prima facie reason for". Semantic inheritance network cannot represent some inference relations that John Pollock's theory can. They also include a principle for prioritizing more specific rules to more general ones.

The AGM theory is a formal theory of *belief revision* developed by Alchourrón, Gärdenfors and Makinson [1, 2]. It tries to answer the question what to do in the case when an agent beliefs a set of propositions and a change (addition or retraction of a proposition) has to be carried out. If a proposition is added which is inconsistent with the original belief set then the agent has to revise the belief set by retracting some beliefs so that the belief set is consistent again. The problem is that logic itself does not provide any reason to prefer removal of one belief over another. Therefore it is necessary to rely on additional information about these propositions. The AGM theory states so called "rationality postulates" which characterize general properties that the revision should conform to. The main idea is that the revision should make a *minimal change* to accommodate the new information consistently.

3.2 Logical approach.

Logical approaches study the non-monotonic consequence relations which are defined on propositions, not on beliefs of an agent. Therefore the focus is not the epistemology, although this approach has implications for epistemology too.

A consequence relation is monotonic iff if a formula follows from a set of formulas then it follows also from all supersets of this set. Any consequence relation which fails this condition is non-monotonic. There are various desirable properties that the non-monotonic/defeasible consequence relation should have and the logical approaches differ in the set of properties that their consequence relation satisfies. In the following, a brief overview of the main logical approaches is given. For a comparison of these techniques see [40].

One of the first defeasible systems was the Reiter's default logic. It is based on the use of default rules. A default rule allows derivation of its consequence only when its prerequisite holds and its justifications are consistent with the current set of beliefs. Classical example of a default rule is the following: $bird(x) : flies(x) \rightarrow flies(x)$ which says that birds typically fly (unless they are for example penguins). We can derive that a specific bird flies unless it is inconsistent with our current set of beliefs. A default theory can have many extensions and even extensions which are mutually inconsistent. Reasoner can usually choose any of these extensions to proceed with inferencing.

Non-monotonic logic I by McDermott and Doyle [47] was developed just after the initial Doyle's truth maintenance system and McDermott and Doyle pointed out their similarity. Non-monotonic logic I uses a modal operator M "possible". So a rule $p \wedge Mq \rightarrow q$ says that if p holds and q is possible then infer q. As in default logic, non-monotonic logic I theory can also have more extensions. Non-monotonic logic I is similar to Autoepistemic logic in that it also uses a "possibility" modal operator and defines extension of a theory similarly. See section 11.1 for more about the relation of autoepistemic logic and truth maintenance.

Circumscription developed by McCarthy uses predicate minimization to express default rules. One or more predicates are selected for minimization and are usually interpreted as abnormality predicates. Models of the circumscription theory are sought which minimize extensions[1] of these predicates. Default rule can then for example be: $P(x) \wedge \neg ab_i(x) \rightarrow Q(x)$ which says if $P(x)$ holds and x is not abnormal with respect to the $i-th$ abnormality ab_i then infer $Q(x)$. McCarthy used circumscription also as an attempt to solve the frame problem.

Other logical approaches include for example preferential logics and logics of extreme probabilities. Circumscription is a special case of preferential logics. Circumscription prefers the models that minimize the extension of the abnormality predicates. Preferential logics use different kinds of preference relations which also are transitive and irreflexive.

This section presented a brief overview of the main defeasible reasoning approaches. Two of them are in the focus of this paper: belief revision and truth maintenance because it is closely related to non-monotonic and autoepistemic logics.

4 Belief revision

Belief revision is a theory formalizing an idea of rational changes to beliefs based on the principle of minimal change (also called informational economy) which says that as much as possible information should be retained when changing ones beliefs. This principle is motivated by the

[1] The extension of a predicate is the set of assignments of values to its arguments that make it true.

assumption that information is precious and unnecessary losses of it should be avoided. Belief revision is also referred to as the AGM theory thanks to its originators Alchourrón, Gärdenfors and Makinson [1, 2]. Belief state is represented by a set of sentences K which is closed under logical consequence. That is $\varphi \in K$ whenever $K \vdash \varphi$. Sometimes the belief state is represented only by a belief base – a set of sentences whose logical closure corresponds to the belief set just defined. This section provides a closer look at belief revision and draws mainly from Gärdenfors's book on Belief revision [28].

There are three kinds of belief changes: expansion, revision and contraction. In expansion a new sentence is added to a belief set K together with the logical consequences of the addition. In revision a new sentence contradicting K is added to K in such a way that the resulting belief set is consistent (so the operation removes some of the old sentences in K). In contraction a sentence is removed from K together with its consequences not derivable from K without the sentence. Belief set is a set of sentences closed under logical consequences. The theory states rationality postulates about revision and contraction operations on a belief set K. The extension operation can easily be defined as the logical closure of $K \cup \{\varphi\}$.

Revision and contraction operations cannot be defined solely based on logical considerations. For example consider a belief set K containing among others also the sentences: p, q, r and $p \wedge q \rightarrow r$. Suppose you want to revise the belief set by adding $\neg r$. Then except r itself, at least one of the sentences $p, q, p \wedge q \rightarrow r$ has to be removed too to maintain consistency. There is no purely logical reason to prefer one of these sentences over another. The rationality postulates formalize an idea what properties should a rational change have.

Lets look at the rationality postulates for the revision operation to give a closer feel of the theory. Revision of a belief set K by a sentence φ is usually designated $K\dot{+}\varphi$ (to distinguish it from a simple extension which is denoted by $K + \varphi$).

($RP1$) For any sentence φ and any belief set K, $K\dot{+}\varphi$ is a belief set.
($RP2$) $\varphi \in K\dot{+}\varphi$.
($RP3$) $K\dot{+}\varphi \subseteq K + \varphi$
($RP4$) If $\neg\varphi \notin K$, then $K + \varphi \subseteq K\dot{+}\varphi$.
($RP5$) $K\dot{+}\varphi = K_{\perp}$ if and only if $\vdash \neg\varphi$.
($RP6$) If $\vdash \varphi \leftrightarrow \psi$, then $K\dot{+}\varphi = K\dot{+}\psi$.
($RP7$) $K\dot{+}\varphi \wedge \psi \subseteq (K\dot{+}\varphi) + \psi$.
($RP8$) If $\neg\psi \notin K\dot{+}\varphi$, then $(K\dot{+}\varphi) + \psi \subseteq K\dot{+}\varphi \wedge \psi$.

($RP1$) requires an update of a belief set to be a belief set. ($RP2$) says that the input sentence has to be accepted. ($RP3$) and ($RP4$) express that revision results in the same belief set as expansion if $\neg\varphi \notin K$. ($RP5$) states that the resulting belief set can only be inconsistent in case that φ is logically impossible. $K_{\perp}$ denotes the inconsistent belief set containing all sentences. There is only one such belief set because whenever K is inconsistent, then $K \vdash \varphi$ for every sentence φ and belief sets are closed under logical consequence. ($RP6$) says that logically equivalent sentences result in identical revisions. Postulates ($RP7$) and ($RP8$) roughly say that subsequent changes should be incremental.

The AGM theory similarly states eight rationality postulates for the contraction operation (denoted $K \dot{-} \varphi$). It can then be shown that the two operations conforming to their respective postulates can be expressed in terms of each other. Revision $K\dot{+}\varphi$ then corresponds to $(K \dot{-} \neg\varphi) + \varphi$ and contraction $K \dot{-} \varphi$ corresponds to $K \cap K\dot{+}\neg\varphi$. Therefore a method for constructing one of the operations would automatically yield construction of the other operation via these definitions. For a specific example how the contraction operation can be modeled see for example the book Belief revision edited by Peter Gärdenfors [28].

Even though the sentences of belief sets are treated as facts and there is no purely logical reason to prefer one over another; in practice some knowledge is more important than other. Belief revision theory reflects this fact by introducing the notion of *epistemic entrenchment*. The degree of epistemic entrenchment has bearing on what is abandoned from a belief set and what is retained. Belief revision states fives postulates for epistemic entrenchment. The reason is to specify a unique belief set for the contraction and revision operations and thus making them into functions. In the following, $\varphi < \psi$ and $\varphi \leq \psi$ represent that "φ is epistemically less, resp. at least as, entrenched as ψ".

$(EE1)$	If $\varphi \leq \psi$ and $\psi \leq \chi$, then $\varphi \leq \chi$.	(transitivity)
$(EE2)$	If $\varphi \vdash \psi$, then $\varphi \leq \psi$.	(dominance)
$(EE3)$	For any φ and $\psi, \varphi \leq \varphi \wedge \psi$ or $\psi \leq \varphi \wedge \psi$.	(conjunctiveness)
$(EE4)$	When $K \neq K_{\perp}, \varphi \notin K$ iff $\varphi \leq \psi$, for all ψ.	(minimality)
$(EE5)$	If $\psi \leq \varphi$ for all ψ, then $\vdash \varphi$.	(maximality)

The postulates specify what is a minimal change in which situation (for example $(EE2)$ says that it is a smaller change to give up φ, if either φ or ψ has to be given up, because if ψ was to be removed then φ would have to be removed too because ψ follows from it). Note that epistemic entrenchment is relative to a belief set - different belief sets have different orderings of epistemic entrenchment. What is gained from defining the epistemic entrenchment postulates is an explicit definition of a contraction function in terms of the ordering $\leq$. This means that the problem of constructing revision and contraction function can be transformed to the problem of providing an ordering of epistemic entrenchment. Gärdenfors discusses also other approaches to constructing revision and contraction functions.

As Gärdenfors points out, belief revision could also be a useful tool to study non-monotonic inferences. There is a connection between belief revisions and the meaning of conditional sentences. Conditional sentence is a sentence of the form "if φ is the case then ψ is the case" or "if φ was the case then ψ would be the case". If the condition φ is in contradiction with the current belief state then the conditional is called a *counterfactual*. It was argued that a formal semantics for counterfactuals is of great value for many problem areas within AI, in particular since they form the core of non-monotonic inferences. The semantics of conditionals can be based on F.P.Ramsey's test: 1. add φ hypothetically to the belief state 2. check if the result leads to a contradiction 3. if yes, then make minimal adjustments to the belief state to restore consistency, but don't change φ 4. consider whether or not ψ can be accepted in this adjusted belief state. Gärdenfors presents further analysis of this test and concludes that the test is inconsistent with $(K\dot{+}4)$. However, if updating (due to Katsuno and Mendelzon) is used instead of revision in formulating the Ramsey test, then this combination is consistent.

From the computational point of view, the problem of belief revision is hard. The problem of deciding whether $\psi \in Cn(K)\dot{+}\varphi$ is already co-NP-complete [49]. Eiter says that this problem amounts to the evaluation of the counterfactual "if p were the case then q would be the case" over K under the respective change semantics and shows that evaluating a counterfactual is for many of the proposed change operators complete for the class Π_2^P, which means that it is unrealistic to expect a polynomial algorithm for evaluating counterfactuals under the respective semantics, even if an algorithm can solve arbitrarily many NP-complete problems at no cost [24]. Eiter shows that the problem is easier if the formulas are Horn clauses. It is also easier when the size of the knowledge change (formula p) is bounded by a constant. But the problem becomes tractable only if both restrictions are considered simultaneously. Eiter concludes that it is therefore important to seek suitable restrictions for the various formalisms to eliminate all sources of intractability. To the best of our knowledge, there still was not done very much in

this direction.

Belief revision seems to lag behind reason maintenance when it comes to practical implementations. This is supported for example by Liberatore's and Schaerf's note that to the best of their knowledge only Winslett in [62] proposed an algorithm for belief revision [41]. In their paper they show relationship between belief revision operators and circumscription and claim that this correlation can be leveraged to a number of existing circumscription algorithms to belief revision. Circumscription is not the only non-monotonic theory that belief revision relates to. Nebel showed a close correspondence of belief revision and default reasoning [49]. Gorogiannis and Ryan also say that implementations of belief revision are scarce but they present implementation of propositional belief revision using Binary Decision Diagrams [32] and apply it in the area of fault diagnosis. As Dixon and Foo point out, the major difficulty in implementing belief revision lies in the fact that it operates on infinite belief sets whereas for practical implementation this has to be changed to some finite representation but in finite representations new problems arise [18].

5 Reason maintenance

The term *reason maintenance* refers to a variety of knowledge base update techniques which are all based on similar design - they are divided into the inference engine and the reason maintenance parts. The original term for these techniques was *truth maintenance* as Jon Doyle named it in his Master thesis and technical report at MIT [19]. Later he however preferred the name reason maintenance for being more precise and less deceptive [21]. Both terms are used in this paper due to the fact that all abbreviations of the names of the algorithms still end with the TMS suffix.

5.1 Introduction

Reason maintenance systems are usually designed to address a problem solving task, such as fault diagnosis, but are applied in other areas too, as it is demonstrated in later sections. The fundamental design choice that all the reason maintenance algorithms share is the division of the problem solver into two components: an inference engine and a TMS (see Figure 10). The inference engine is usually first order, operates on a set of rules representing the domain knowledge and communicates the inferences to the TMS. Responsibility of the TMS is to determine what facts are believed and which are not believed given the information received from the inference engine. The TMS records and uses this information in form of justifications (see Section 7 for details).

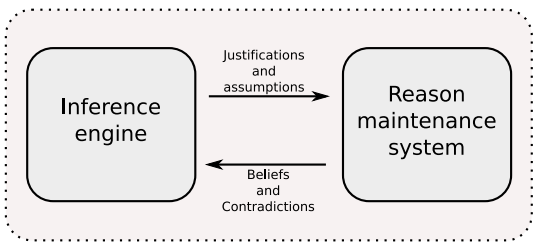

Figure 10: Two components of a problem solver.

Forbus [27] lists four problem solver issues that justifications facilitate:

- Problem solver can generate explanations by tracing justifications for a belief.
- Sources of a wrong conclusion or a contradiction can be traced via justifications.
- Justification provide a cache of inferences and keep track of which worked and which did not.
- Problem solver can do default reasoning because justifications contain assumptions explicitly.

Reason maintenance systems provide a general framework for building problem solvers but, as Forbus points out, they are not always appropriate. One of the crucial features of a TMS is the justification caching. Some inferences may be computationally very expensive and therefore the problem solver should avoid recomputing such inferences, which is exactly where the cached justifications help to improve performance. On the other hand, if running rules is inexpensive, then the performance advantage of justification caching is lost and the memory overhead of keeping always all justifications may become unjustified. This problem has however been already partially addressed in the newer, modified, TMS algorithms described in later sections.

5.2 TMS algorithms overview

In the next sections, the basic TMS algorithms as well as their modifications are described:

Mon. JTMS	Monotonic justification based TMS - a simplification of the original non-monotonic JTMS.
Nonmon. JTMS	The original non-monotonic, justification based TMS.
LTMS	Logic TMS - a JTMS based TMS modified to to accept arbitrary clauses.
ATMS	Assumption based TMS - de Kleer's TMS designed to solve some of JTMS's shortcomings.
HTMS	Hybrid TMS - by de Kleer, combines advantages of JTMS/LTMS and ATMS.
ITMS	Incremental TMS - designed to improve LTMS context switch performance.

Each of these algorithms has its own problem domain where it performs the best. An exception is perhaps the HTMS which combines features of the JTMS and ATMS and behaves as the former or the latter depending on its usage. Note that, unlike belief revision, reason maintenance is more pragmatic; it started with practical implementations and formal description and theory came only later.

6 Preliminaries

This section introduces definitions of terms needed and used in later sections.

Literal If A is an atom then both A and $\neg A$ are literals. Atom is a symbol in logic that can take the value either true or false.

Clause A clause is a disjunction of finitely many literals.

Horn clause Horn clause is a clause that contains at most one positive literal. It is usually written as $L_1 \vee \ldots \vee L_n \rightarrow L$, where $n \geq 0$ and L is the only positive literal.

Definite clause Definite clause is a Horn clause that has exactly one positive literal.

Unit clause Unit clause is a clause containing only one literal. A set of clauses that contains a unit clause l can be simplified by removing each clause containing l and by deleting $\neg l$ in each clause that contains it.

Graph Graph G is an ordered pair $G = (V, E)$ where V is a set of nodes and E is a set of edges. In an undirected graph, edge is a set of two nodes. In a directed graph, edge is an ordered pair of nodes. The number of nodes and number of edges are denoted $|V|$ and $|E|$ respectively.

Connected graph Let $G = (V, E)$ be an undirected graph. Two vertices $u \in V, v \in V$ are called connected if G contains a path from u to v. G is called connected if every pair of distinct nodes in the graph is connected.

Component Connected component, or a component, of an undirected graph is is a maximal connected subgraph of the graph.

Strongly connected component Strongly connected component is maximal subgraph of a directed graph such that for every pair of vertices u, v in the subgraph, there is a directed path from u to v and a directed path from v to u.

Power set Power set of a set S, designated $P(S)$, is the set of all subsets of S.

7 JTMS

JTMS - justification-based TMS uses so called justifications as a means of tracking dependencies between inferences made by the reasoner. It is the first TMS that was proposed in 1978 by Jon Doyle in [19]. Subsequently it has been studied and refined and the other reason maintenance systems, ATMS, LTMS and HTMS (and their variations) are inspired by it. Doyle's TMS is non-monotonic which means that it supports rules such as "if switchClosed unless bulbBroken then light". We will however introduce monotonic JTMS first because it is simpler.

A monotonic TMS is a general facility for manipulating Boolean constraints on proposition symbols [45]. In our terminology we call constraints rules. For example a rule could say that pages tagged as "bug" and "new" are tagged as "todo". This rule would have the form $P \wedge Q \rightarrow R$ where P, Q and R are some proposition symbols interpreted by the outside world. P could stand for "bug" and the outside interpretation could be "this page describes a bug". In the basic JTMS only definite clauses are allowed as rules.

The name "justification-based" TMS results from the way TMS maintains knowledge. For each proposition it creates a node and connects the nodes by justifications which are (instances of) rules. Justifications express dependencies between nodes which are then used to update the graph of nodes when the problem solver communicates a change to the TMS. Nodes that are justified by justifications with no antecedent are called premises. Given a set of rules and a set of propositions TMS can be asked about their consequences.

Generic TMS interface Problem solver communicates new inferences to the TMS and can ask the TMS about the current state of knowledge. The communication between problem solvers and different TMS systems is quite similar and was generalized by McAllester who introduced a generic interface of four different functions [45]. The first interface function, *add-rule*, adds a rule to the internal set of rules. The further functions are used to query the TMS about the state of the knowledge: *follows-from*, *justifying-literals* and *justifying-rules*.

The *follows-from* function takes two arguments - a literal φ and a set of literals Σ. The function call *follows-from*(φ, Σ) can return three different values: yes, no, or unknown (if only definite clauses are allowed then yes and no values suffice). The yes and no values are self-explanatory - it is guaranteed that φ follows (resp. does not follow) from the premise set Σ and the set of internal rules. If the function returns *unknown* then the system was not able to determine if φ follows although the right yes or no answer would be computable with corresponding effort [6]. In other words, the TMS answers unknown if it cannot decide between the two cases efficiently.

If TMS answered yes to a *follows-from*(φ, Σ) question then one can ask the TMS to justify this answer. This is the purpose of the third and fourth function. φ follows from literals returned by *justifying-literals*(φ, Σ) and rules returned by *justifying-rules*(φ, Σ). Note that using these functions recursively, justification tree for a derived fact can be created.

Consider the rules $P \to Q, (P \wedge W) \to R$, and $(Q \wedge R) \to S$. Most TMS algorithms are able to derive S from these rules and the set of premises P, W [45]. To be able to do this derivation the *follows-from* function has to be implemented as transitive [6]. Besides transitivity the *follows-from* function is usually implemented to be also reflective and monotonous.

Once a rule is added it can never be removed. Also, justifications (and contradictions) cannot be retracted. They form a cache of the problem solver - once something has been derived it will be present so it does not have to be re-derived.

Contradiction handling. The basic JTMS cannot directly detect contradictions because it is restricted to definite clauses. That is, it can represent literal φ but cannot represent literal $\neg\varphi$ because that is a non-definite singleton clause. Contradiction can be represented as a special node. Then whenever there are two nodes contradicting the inferencer has to inform the TMS about the contradiction by supplying a justification with the contradicting nodes in antecedent and the special contradiction node in consequent. Or, to conform with the TMS interface functions, rules with $\bot$ in the head can be allowed. It can be beneficial to create a new contradiction node for each contradiction as it can facilitate debugging and explanation [27, 9].

We have not yet introduced a TMS interface function that would inform the inferencer about contradictions. McAllester proposes adding a special propositional symbol called *contradictory* and using it in the follows-from function: *follows-from*$(\textit{contradictory}, \Sigma)$. Beckstein proposes using a separate function *contradictory*(Σ) which then can be used to define McAllester's follows-from function in the case when the first argument is the special symbol *contradictory*. Justification functions can be used to find the root cause of the contradiction.

Implementation Because only definite clauses are allowed the *follows-from* function (and therefore most of the basic JTMS) can be implemented as a simple forward propagation. Contradictions do not pose a problem because contradiction is only a node with the property that if it can be derived then the inferencer must be informed. If more general clauses were allowed then a more complex algorithm would be needed.

Well-foundedness To be efficient, TMS algorithms try to examine and update as little of the graph as possible. However, if too little is examined then it can lead into problems with circular justifications, which is called the problem of well-foundedness [15]. TMS algorithm must be carefully designed and implemented to maintain only well-founded, non-circular, justifications. To illustrate the notion an example based on a example by de Kleer [15] follows. Suppose there are three nodes: "Bug", "New" and "Todo". The node "Bug" is a premise, it is always labeled TRUE. Inferencer uses two rules: $Bug \wedge New \rightarrow Todo$ (R_1) and $Bug \wedge Todo \rightarrow New$ (R_2). Suppose that the system is given justification J for node "New". Then, using rule R_1, the inferencer infers "Todo" and TMS adds a justification for it to the graph, see Figure 11.

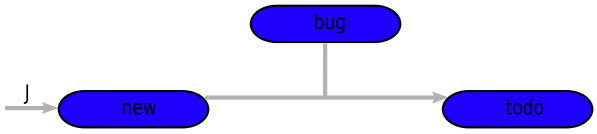

Figure 11: Dependency graph consisting of three nodes and two justifications.

Then, suppose that for some reason J is no longer valid and the system is given a new justification K for the node "Todo". In this state the inferencer derives "New" using rule R_2 and TMS adds a new justification for this derivation to the graph, see Figure 12. Problem can arise if K becomes invalidated in this state. Because justifications are never removed from the dependency graph, justification J is still present in the graph. Therefore "Todo" could be supported by it which would in turn support "New". This support would however not be well-founded because it is circular.

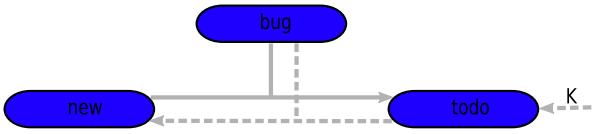

Figure 12: Circular dependency violates the well-foundedness requirement.

JTMS shortcomings JTMS is logically very weak because it allows only definite clauses. It cannot directly represent that $\neg P$ is negation of P. It can do it only indirectly by establishing two nodes, let's say P and $nonP$, and a rule saying that these two nodes are in contradiction with each other: $P \wedge nonP \rightarrow \bot$. Note that for JTMS $nonP$ is only a name of a variable, not a negation of P, it is the problem solver who interprets it as negated P.

Encoding arbitrary clauses in JTMS is possible but requires expansion. Suppose we would like to encode the clause $A \vee B \vee C$. That would require creating six nodes: $A, nonA, B, nonB, C$ and $nonC$ and a set of definite clauses:

$$\begin{array}{ccc} A \wedge nonA \rightarrow \bot & B \wedge nonB \rightarrow \bot & C \wedge nonC \rightarrow \bot \\ nonA \wedge nonB \rightarrow C & nonA \wedge nonC \rightarrow B & nonB \wedge nonC \rightarrow A \end{array}$$

Note that similar expansion is necessary for all clauses: to encode $A \rightarrow B$ one has to encode the rule $nonB \rightarrow nonA$ too. Without the contrapositive rule the JTMS would not be able

to label A false if B was labeled false. It is also important for the efficiency of backtracking. Suppose that $A_1, A_2, ..., A_n$ underlie a contradiction. That means that the clause $\neg A_1 \vee \neg A_2 \vee ... \vee \neg A_n$ is true. But JTMS cannot represent it directly. If backtracking uses fixed order and explores A_n as the last then it is only necessary to add two rules to the JTMS to represent this contradiction: $A_1 \wedge A_2 \wedge ... \wedge A_{(n-1)} \rightarrow nonA_n$ and $A_n \wedge nonA_n \rightarrow \bot$ [27].

These examples show that a greater expressivity in the TMS is desirable. That is the reason why the LTMS, a logic-based TMS, was devised.

8 LTMS

The LTMS, Logical Truth Maintenance System, was first developed by McAllester in [43, 44]. LTMS is more powerful than JTMS because it can represent general formulas as justifications and it can also represent negated nodes. On the other hand it also means that its implementation requires a more sophisticated algorithm.

The implementation of the *follows-from* function is in the LTMS usually done with a conceptually simple procedure known as *Boolean constraint propagation*, or BCP. Boolean constraint propagation is similar to the Davis-Putnam algorithm [44, 10] and was shown to be equivalent to unit propagation, a form of resolution for propositional logic, in the sense that each method can be simulated by another in constant time [5]. In LTMS, justifications are represented as general disjunctive clauses. It can be assumed that the set of rules is in conjunctive normal form (CNF). The nodes can be labeled with one of three different labels: YES, NO, UNKNOWN. Initially all nodes are labeled UNKNOWN. To compute the consequences of a particular premise set Σ one assigns the label YES or NO to each proposition symbol in the set of literals Σ depending on whether that symbol appears positively or negatively in Σ. New labels are then computed based on local propagation - whenever a new truth label follows from existing labels and some single rule, that new label is added to the network and propagation continues. If a set of derived labels ever violates one of the internal constraints, then the special proposition *contradiction* is labeled true [45]. This propagation process can be run to completion in time linear in the total size of the set of rules (clauses) [44]. The *follows-from*(φ, Σ) function simply runs the BCP algorithm starting with the labels in Σ and determines if a label is derived for the proposition symbol in the literal φ. If a label has been derived and its sign is the same as the sign of the literal φ, then *follows-from* returns YES, otherwise returns UNKNOWN. Discussion of implementation of the rest of the functions can be found in [6].

Boolean constraint propagation is not logically complete. For example the literal Q follows from the rules $P \rightarrow Q$ and $\neg P \rightarrow Q$ but does not follow from either rule individually. Therefore BCP will not infer Q even though Q follows from the rules. The incompleteness of BCP represents a compromise between functionality and efficiency [45] and amounts to a constructive bias. Boolean entailment is coNP-complete so no efficient and complete algorithm can be expected.

Making BCP complete Although BCP is generally incomplete, there is a way to make it complete and even to control the degree of completeness. The technique exploits an observation that if the set of propositional formulae are converted to their prime implicates (definition follows shortly) then BCP is complete. This approach is described for example by de Kleer in [13]. He observes that many problems, mainly those which need reasoning about physical world, are local in the sense that each constituent of the problem has a fixed behavioral model.

Therefore much of the reasoning can be viewed as propagation inside the different models independently. This observation leads him to the idea to split all formulas describing the whole problem into separate modules which can then be made complete separately using prime implicates. TMS can then use a complete inference procedure locally on modules. As a result the desired functionality can be achieved without incurring substantial performance degradation. This also enables a dynamic trade-off between completeness and efficiency - to achieve more efficiency more modules can be created, to achieve more completeness the modules can be merged. If every formula is regarded as a separate module then the algorithm is equivalent to the traditional BCP. This technique can be applied to the ATMS too.

De Kleer defines prime implicates as follows. Clause A is subsumed by clause B if all the literals of B appear in A. Then, an implicate of a set of propositional formulae F is a clause entailed by F not containing complementary literals. A prime implicate of a set of formulae F is an implicate of F whose no proper subclause is an implicate of F. For example the conjunction of clauses $\neg x \vee y \vee z$, $x \vee y \vee z$ has one prime implicate: $y \vee z$. But usually there is more prime implicates than the conjuncts in the CNF of the formula [13]. Consider the following example:

$$\neg a \vee b,\ \neg c \vee d,\ \neg c \vee e,\ \neg b \vee \neg d \vee \neg e.$$

These all clauses are prime implicates but there are three more:

$$\neg a \vee \neg d \vee \neg e,\ \neg b \vee \neg c,\ \neg a \vee \neg c.$$

This example illustrates that the set of prime implicates required to make BCP logically complete can be extremely large. Therefore it is impractical to exploit this technique directly. De Kleer points to a number of papers describing a variety of different algorithms for computing prime implicates. He proves four theorems which say that BCP can be made complete if needed and that running BCP on the prime implicates of the individual formulae is the same as running BCP on the formulae. Therefore efficient implementations of clausal BCP can be used. Because direct use of the algorithm is too expensive, de Kleer discusses several techniques to make it more efficient. The intuition behind these techniques is that because constructing and storing implicates is the most expensive part of the algorithm, it should be delayed as much as possible. De Kleer implemented his algorithm in both LTMS and ATMS and used it in his Qualitative Process Engine, QPE, to solve qualitative physics problems.

Single-context TMS JTMS and LTMS are single-context operating algorithms. In each state they manage only one current context where context is the current set of premises and assumptions. The premises and assumptions describe the current trial to solve the problem at hand. Problem solvers search the state space by switching the current context. When a context switch occurs then traditionally JTMS and LTMS will remove all labels, update the set of premises and assumptions and start the labeling process again. In general problem solving computing a label of a node may require a costly computation or even calling an external program. At the same time context switches are likely to occur often. That means that for many problem solving applications JTMS and LTMS can be inefficient. Therefore new TMS systems capable of managing multiple contexts were created.

9 ATMS

Many problem-solving tasks require the inference engine to rapidly switch among contexts or to work in multiple contexts at once. Two examples of such tasks are qualitative reasoning and

diagnosis [27]. Switching context in JTMS and LTMS requires relabeling nodes which can be costly and it is the main reason for developing and using ATMS. There are other problems which de Kleer discusses in [12] but these are mostly related to properties of the original, Doyle's, non-monotonic JTMS. In [12] he also analyzes the systems of Doyle [20], Martins [42], McAllester [44], McDermott [46] and Williams [61] and points out how these TMSs have addressed or failed to address these problems. Assumption based TMS [11] is similar to the monotonic JTMS but it avoids such relabeling. In the ATMS implementation the propagation process is independent of any particular premise set - a single "universal propagation process" pre-computes all answers to all possible queries [11]. To make the universal propagation process more efficient, the user can declare an a priori set of "possible premises". Each possible premise is a literal and every premise set in every query to the ATMS must be a subset of the set of possible premises.

ATMS context switches are free because the labeling was already precomputed in form of complex *labels* for each node. Roughly speaking, this label describes different sets of assumptions under which the node would be labeled YES in JTMS or LTMS. An ATMS label is a set of environments where an ATMS *environment* is a set (a conjunction) of assumptions. A node holds in an environment if it can be derived from the set of assumptions and the set of justifications. A *nogood* is a contradictory environment. Environment is consistent if it is not a nogood. A context of an environment is the set of nodes that hold in the environment. So while a justification describes how a node depends on its antecedents, an ATMS label describes how the node ultimately depends on assumptions.

ATMS algorithm guarantees that each label of each node is consistent, sound, complete and minimal. A label is consistent if all its environments are consistent; it is sound if the node is derivable from each environment of the label; it is complete if every consistent environment in which the node is derivable is a superset of some enviroment of the node's label. The label is minimal if there are no two environments in the node's label such that one is the superset of the other. When a new justification for a node is introduced by a problem solver, an incremental change of the node's label is evaluated using the labels of the node's antecedents. After this incremental change is made consistent and minimal, it is combined with the node's old label to make a new minimal node's label. Note that once derived, a node is never re-derived in a new environment. Only its label has to be changed [56].

Consistency JTMS and LTMS insist on consistency in the sense that the current set of premises and assumptions must be consistent with the rules. There are two reasons for this according to [11]: 1) the problem solver operates with a single controlled focus, 2) the presence of inconsistency renders all deductions meaningless. De Kleer argues that only assertions directly affected by the contradiction should be retracted. The fundamental difficulty is that it is costly to determine which data are affected. In an assumption-based TMS, one can tell directly whether a node is affected or not, so demanding consistency is unnecessary. A contradiction merely indicates that a specific set of assumptions is inconsistent, and that data depending on them are affected.

When a nogood, a contradictory environment, is detected, it is added to a nogood database. This environment and its supersets are then removed from all node labels. The nogood database is kept minimal to prevent examination of environments known to be inconsistent [56].

Complexity If there are n assumptions then there are potentially 2^n contexts (recall that context is the current set of premises and assumptions). There are $\binom{n}{k}$ environments having k assumptions. De Kleer says that ATMS is practical even when n is as large as 1000 [11].

The ATMS is then exploring a space of size 2^{1000}. There are two observations about contexts that help to explain the efficiency. First is the fact that a node is in every superset context of its context as well so it is only necessary to record the greatest lower bound environments in which a node holds. Similarly, it is only necessary to record the greatest lower bounds of the inconsistent environments. Second, in cases where most of the environments are inconsistent, these inconsistencies are identifiable in small subsets of assumptions. Therefore, large parts of the environment lattice need never be explicitly checked for consistency [11].

The worst case behaviour, when ATMS implementation requires both exponential time and exponential space to answer a single query, can however be easily realized even in simple cases [45]. ATMS was created to support problem solvers interested in all the solutions of a problem and in this case computing all the solutions by ATMS can be more efficient than all the context switching that would be needed by LTMS or JTMS algorithms.

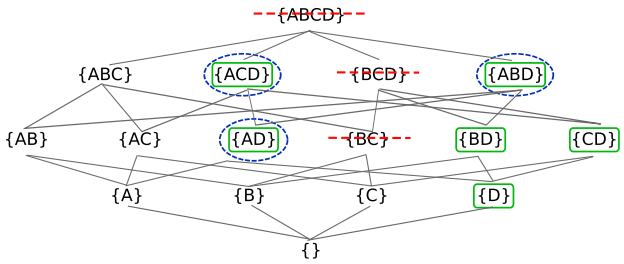

Figure 13: Example of an ATMS environment lattice.

Example To get a better idea of the notions presented in this section consider Figure 13 which depicts an environment lattice for assumptions A, B, C, and D. Recall that an environment is a set of assumptions and there are $\binom{n}{k}$ environments having k assumptions. The lines between environments represent a subset relationship. From the picture it is easily seen that the top-most environment is a superset of all environments which in turn all contain the empty environment as a subset. Inconsistent environments (environments from which it is possible to derive $\bot$) are crossed out in the figure and correspond to nogoods. In this case they are result of a single minimal nogood $\{B, C\}$. In ATMS, each fact has a set of environments in which it holds. In the example in Figure 13, the squared environments denote all environments of the fact $needsRevision(123)$ (where 123 is a unique identifier of a specific page). The circled environments represent environments of the fact $reviewer(123, john)$. When the inferencer deduces a new fact, $todo(123, john)$, (according to rule R_4 of Scenario 1 in Section 2.1) then its set of environments can be determined as the intersection of sets of environments of the first two facts. The environments from this intersection are both circled and squared in the figure. The label of the fact $needsRevision(123)$ is the set of greatest lower bounds of the squared environments: $\{\{D\}\}$. The label of the fact $reviewer(123, john)$ is the set of greatest lower bounds of the circled environments: $\{\{A, D\}\}$. And the label of the derived fact, $todo(123, john)$ is the set of greatest lower bounds of the both circled and squared environments: $\{\{A, D\}\}$. In this simple case each of the labels contains only one environment but it is easy to think of a bigger example where the labels would contain more environments. For a bigger example see [11].

10 TMS modifications

It was soon realized that many applications, including diagnostic ones [16, 17] only consider a small number of the possible contexts so the ATMS algorithms waste effort exploring contexts irrelevant to the diagnostic goal [14]. On the other hand, there is usually too much context switching to use a single-context TMS effectively. This observation led to development of alternative TMS algorithms which try to focus the ATMS to a smaller set of contexts. One example is de Kleer's HTMS - Hybrid Truth Maintenance System [14]. In other applications such as operating systems of embedded devices even a single context LTMS can be too inefficient due too unoptimal context switching algorithm and space requirements as justifications can only be added, not removed. This problem is addressed in several proposals (i.e. [26]) how to improve the performance of these algorithms, resulting for example Nayak's and Williams's ITMS - Incremental Truth Maintenance System [48] and its further refinements [63].

10.1 HTMS

Hybrid Truth Maintenance System was developed by de Kleer [14] in response to the inefficiency of ATMS in diagnostic applications. Although diagnostic applications require working in multiple contexts they rarely need to use all of them and hence the inefficiency. De Kleer also points out that the idea of a generic TMS interface suggests that JTMS, LTMS or ATMS are better thought of as algorithms with differing time-space tradeoffs. Therefore they should be rather referred to as JTMS algorithm, LTMS algorithm and ATMS algorithm. The HTMS algorithm also has the familiar interface, cf. Section 7, but adapts to the task as problem solving unfolds.

If the inference engine remains within a single context, then the HTMS behaves like a conventional JTMS or LTMS and if the inference engine explores all contexts, then the HTMS behaves like an ATMS. The inference engine can control the computation of the HTMS to only focus on gathering the most useful information which is relevant to answering queris about the contexts of interest. De Kleer reports a greatly improved efficiency [14]. For example one task that did not end after 240 hours of computation using the ATMS was completed in less than a second with the HTMS. It is important to note that, to exploit the HTMS, problem solvers have to be redesigned to utilize its features fully.

The HTMS algorithm can be viewed as an ATMS algorithm with three modifications:

- The HTMS only maintains labels with respect to a specified set of context - each context characterized by a focus environment.
- For every focus environment, the HTMS finds at most one label environment for each TMS node.
- The HTMS incorporates a scoring function which guides the HTMS to find the single 'best' label environment for each node.

The first modification avoids much of the combinatorial explosion encountered using the ATMS as irrelevant label environments lying outside of the focus environments are not maintained. The second modification eliminates the combinatorial explosion where a node has an exponential number of derivations within the same context. The incorporation of a scoring function is important in backtracking where one would like to find the smallest nogood supporting the current contradiction. While the ATMS always finds the smallest nogood because

it finds all of them, the HTMS can be directed to find small ones. The function could also compute a score based on the probabilities of the assumptions.

The major interface difference between a single-context TMS algorithm and an HTMS algorithm is that if the set of premises is found inconsistent with the set of rules, then the HTMS is expected to return a constraining nogood which it will find also using the scoring function. The major interface difference between an ATMS and an HTMS is that the label it maintains for nodes is incomplete - for each node it maintains a single, low scoring, supporting environment within the focus. The HTMS algorithm is designed to ensure efficient context switches and is based on a basic intuition: only compute enough of the ATMS labels that are necessary to find low-scoring nogoods and answer queries. Also, even though there mey be a large number of possible label environments for a given set of premises, the HTMS adds only the best environment to each label.

The HTMS algorithm is based on the incomplete but efficient BCP algorithm. De Kleer discusses [14] possibilities to make the algorithm complete but rejects them for being too inefficient. He also gives a comparison of the HTMS algorithm to optimal which shows that the BCP-based HTMS algorithm is nearly optimal for some of the most common ATMS application.

The core of the HTMS algorithm is a best-first propagator which propagates environments with best scores first. It constantly computes the supporting environments for nodes and caches the results. When it is asked for a support of a node it first checks the cache and, only if this is unsuccessful, it tries to find a lowest scoring in focus environment in the node's label. For a detailed description of the algorithms please refer to [14]. The worst case complexity of the HTMS algorithm is $O(mac)$ where m is the number of literal occurrences in justifications (if there are two justifications one with 2 antecedents and another with 3, then $m = 5$), a is the maximum number of assumptions employed in a task and c is the number of contexts explored [14]. This reveals the hybrid nature of the HTMS - c may approach 2^a if the number of contexts considered are large. On the other hand, if the number of contexts is small, then the complexity of the HTMS is linear as that of LTMS.

10.2 ITMS

One inefficiency of LTMS that has already been mentioned is that of costly context switches. When making a context switch, the LTMS algorithm takes a conservative approach and relabels all nodes regardless of whether it is actually necessary. This observation and the need to build a highly efficient TMS system for real-time use in embedded devices led to the development of the Incremental Truth Maintenance System, ITMS [48].

NASA's Nayak and Williams needed to develop an embedded real-time execution kernel, called Livingstone [60], that performs a variety of functions such as commanding, monitoring, diagnosis, recovery and safe shutdown automatically with required response times on the order of hundreds of milliseconds. They report that LTMS can spend a significant percentage of its time working on labels that remain constant between contexts which leaves much room for improvement. On a real-world spacecraft control problem the overhead was 37% on average and rose to about 670% in the worst case. The ITMS algorithm reduces the overhead approximately seven times on the spacecraft problem as reported in [48]. The algorithm finds quickly alternate supports for propositions while guaranteeing well-foundedness and provides a mechanism for propagating the consequences of newly added clauses before other clauses are deleted, increasing the number of consequences available to be used as alternate support.

Please note, that this modified algorithm actually addresses the problem highlighted in

"Change 2" of the motivating scenario from Section 2.1. The change shows that implementing a change by deleting a fact followed by adding a new fact is not optimal because this way it is not possible to preserve propagations that hold both before and after the context switch but that do not hold in the intermediate context. The problem is that the LTMS algorithm does not first look for ways to resupport nodes before removing their label. But even if it looked for resupport first it wouldn't find it in the "Change 2" problem. It would find it if the componentSpecification tag was added first and only then the featureSpecification tag was removed. This however may bring another problem because these two tags may be mutually inconsistent (e.g., a page can have only one specification type). This inconsistency is a barrier to propagation and overcoming this barrier requires an algorithm for propagating through a conflict, which the ITMS algorithm provides.

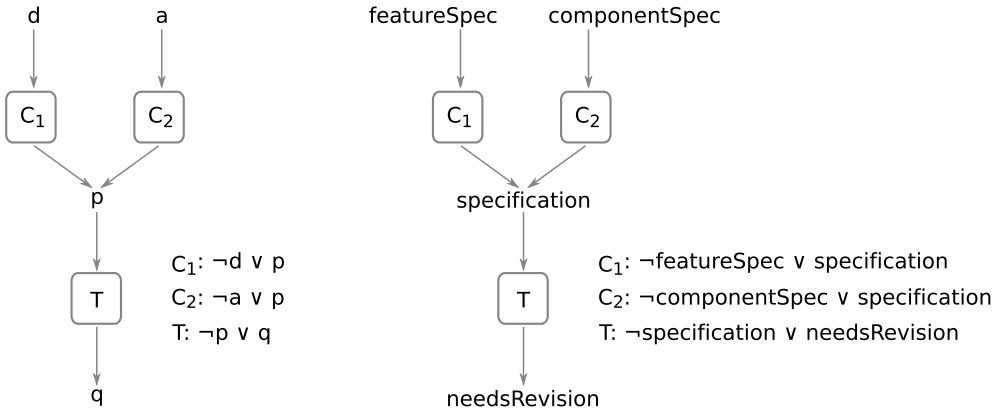

Figure 14: Example of the ITMS resupport problem.

Consider an example in Figure 14 (inspired by an example from [63]). To show the parallel with the "Change 2" both trees are shown but only the first is referred to in the following text. Assumption d is to be deleted and assumption a is to be added. Assumption a satisfies clause C_1 and assumption d satisfies clause C_2. C_1 deduces p logically before d is retracted. When a is added, C_2 becomes unit-open and deduces p. In a LTMS p would first loose support from C_1 and then get support from C_2 as d is deleted and a is added. In order to resupport p immediately, the following condition must be enforced: *d has no influence on the literals in C_2 other than p.* If d influences another literal other than p in C_2, then d can influence two propositions, which will all be labeled UNKNOWN when d is retracted. It means that C_2 can not become unit-open and therefore can not support p in the new context. It is, however, expensive guaranteeing this condition because it may involve a complete traversal of the dependency graph. Therefore Nayak and Williams make the approximation that d does not affect other literals in C_2 and specify under which conditions p can be resupported:

- If p occurs positively (negatively) in clause C_1 then it occurs positively (negatively) in C_2. This makes sure that C_2 would resupport p while preserving its label.

- All literals in C_2 other than p are negative. This ensures that C_2 can propagate a label to p.

- None of the other literals in C_2 depend on p so that C_2 can provide p with a well-founded support.

Ensuring the third condition can be time consuming. Therefore Nayak and Williams proposed a fast approximation that is sufficient for the third condition to hold. Their idea is to associate a *propagation number* with each node. The propagation number of a node is equal to the level at which the node is located in the dependency graph. This allows to replace the third condition with the following one:

Propagation-number-invariant The prior propagation number of p is greater than the maximum of all the other propagation numbers of literals appearing in C_2.

This condition is sufficient but not necessary for the original one and therefore the algorithm may miss resupport opportunities as Williams and Nayak point out [48]. Another problem occurs when C_1 and C_2 imply different labels for p. Nayak and Williams use an approximation in this case too. Propagating through conflict then proceeds as follows:

- Change the label of p from TRUE to FALSE or vice versa. Change the propagation number of p to a value greater than the maximum of propagation numbers of other propositions appearing in C_2.
- Resupport any propositions that can be inferred by using p with its new label if p appears in the clause used for resupport.
- Undo propagations based on the previous label of p.

The above "Propagation-number-invariant" guarantees that the algorithms for resupporting propositions and propagating through conflicts yield well-founded supports.

Nayak's and Williams's experimental results show that on the spacecraft problem the average performance of the ITMS is only 5% off ideal with worst case overhead of 100%.

10.3 Improved ITMS

The ITMS, although an improvement over LTMS, works only based on approximations which means that it can miss some opportunities for resupport. This problem was adressed by Guofu Wu and George Coghill in their Propositional Root Antecedent ITMS [63]. Their method is based on *root antecedents*, which are those antecedents with zero propagation numbers, and can determine the dependency relationship between a node and the rest of the clause accurately and efficiently.

Thanks to root antecedents Wu and Coghill do not have to approximate how d influences C_2 in order to find resupport for p, see Figure 14. If d has an influence on C_2, the root antecedents of the influenced literal in C_2 must contain the root antecedents of d. Then the conditions for p to be resupported are:

- p has to have the same linked label in C_1 and C_2 so that it can keep the same label in the new context.
- all other literals in C_2 other than p are negative.
- p is influenced by assumption d.

- d influences no other literals in C_2 than p.

Their algorithm for finding resupport of p when d is removed and a added is:

- Change p's support from C_1 to C_2. Remove d from the root antecedents of p and add a.
- Resupport nodes that previously depended on p: remove d from their root antecedents and add a.
- Remove d.

If p does not have the same labels in C_1 and C_2 then the ITMS must propagate through a conflict. Thanks to Wu and Coghill's root antecedents this can be done precisely and efficiently. Their conditions for propagation through a conflict are:

- p has different labels in C_1 and C_2.
- d has influence only on p in C_2.

The algorithm for propagation through conflict then is:

- Relabel p from TRUE to FALSE or vice versa. From the root antecedents of p remove d and add a. Change support of p to C_2 (from C_1).
- Set consequence of C_2 to p and consequences of C_1 to none.
- Push all clauses of p to a stack and continue propagation based on the new label of p.
- When the propagation stops in a terminal node, d is removed using the LTMS algorithm.

The approximative nature of the original ITMS makes it efficient in many cases but it is not guaranteed that the propagation will always work so in some cases ITMS may require even more work than the LTMS. ITMS based on root antecedents does a little more propagation than Nayak's and Williams's ITMS but the propagation is reasonably fast because the dependency path was already identified and the labels of all the nodes along the propagation path are kept the same. And the root antecedent ITMS works precisely [48]. Another nice feature of the improved ITMS is that it can provide explanations immediately without propagation or search because every node is already associated with their root antecedents. It is useful to note that the intended application of Wu and Coghill's ITMS is somewhat different from the one of Nayak and Williams. Wu and Coghill integrated the ITMS into a model based diagnosis engine for a chemical plant.

10.4 Fact Garbage Collection for LTMS

In the usual TMSs the size of dependency network grows monotonically as new assumptions are being added and retracted. This growth may become a limiting factor in applications where memory is precious. In one fielded application described by Everett and Forbus [26] the time taken to revise an assumption increased more than fifteen times by the thirtieth change and by 48th change it filled all available memory and crashed. Their solution to the problem is to create a fact garbage collector which removes information unlikely to be useful in the future.

The core of the idea is to identify a class of facts that:

- Make up a significant portion of all the facts.
- Are unlikely to become valid again once they are retracted.
- Are cheap to rederive.

Deleting these facts helps to maintain size of the dependency network but at the same time assures that possible necessary rederivations are cheap. They applied this Fact-Garbage-Collecting technique, a term coined by Stallman and Sussman [55], in an intelligent learning anvironment called CyclePad which is a system used for learning about engineering thermodynamics. Students using CyclePad change parameters of the simulated system and this way they develop better intuitions about thermodynamics. In their system one change can have extensive consequences as a typical cycle requires between 10 and 30 assumptions and may result in derivation of hundreds of additional numerical values. This fact together with the monotonical growth of TMS dependency network result in slow computation times. To alleviate the problem they identified numerical assumptions as facts which fit the three criteria described above and therefore are good candidates for removal. Using their algorithm removing these facts they were able to make 1000 consequtive retractions and assumptions with significant time improvements and virtually no growth in data structures.

The algorithm adds an additional step to the normal LTMS retraction algorithm which is the garbage collection run on facts labeled as UNKNOWN. In this step, for each node it is checked whether it is garbage-collectable and if yes it is collected. Colletion of a nodes involves checking for a few conditions that assure that no information is lost if rederivation is needed. This is necessary in cases where one clause contains both collectable and non-collectable nodes and it involves creating an auxiliary data structure. This data structure is then used to inform the inferencer that some additional step is needed to rederive a previously retracted fact.

Everett and Forbus developed this improvement for the LTMS algorithm but they suppose that all of the concepts are directly applicable to JTMS algorithms too. It is not so clear whether some equivalent improvemnt would be possible for the ATMS. They also analyzed the complexity of the algorithm and found it to be linear in the size of the LTMS dependency network. In best case there also will be no growth in the size of the dependency network, no matter how many assume/retract cycles are run. On the other hand, in many cases there still will be growth because non-collectible facts can be introduced as a consequence of a collectible assumption and such facts are never destroyed. They expect their algorithm to be valuable for applications which use TMS to provide explanations rather than to guide search.

11 Non-monotonic JTMS

The Doyle's original JTMS is non-monotonic and therefore it infers not only based on what information is present but also on what information is not present. In other words, it supports rules like R_3 from Section 2.1: $specification(x) \wedge \neg revised(x) \rightarrow needsRevision(x)$. That is, for *needsRevision* to be supported there must be an evidence of *specification* and no evidence of *revised*. If an evidence of *revised* appears then *needsRevision* has to be invalidated. This JTMS uses a different labeling from the previous ones. A node that is believed (given rules, assumptions and premises) is labeled as IN, otherwise it is labeled as OUT. So node for *needsRevision* would be labeled IN if the node for *specification* was labeled IN and the node for *revised* was labeled OUT. Doyle's JTMS supports non-monotonic inference: if later *revised*

is inferred then *revised* is labeled OUT and together with it also all consequences depending on it. It also handles contradiction: if $\neg needsRevision$ is introduced then it finds the source of the contradiction and tries to resolve it by using dependency directed backtracking, DDB. DDB then may create justification for *revised* to resolve the contradiction. This TMS is a propositional system - nodes represent propositions and the TMS does not understand their internal structure. In this respect this JTMS is similar to the monotonic JTMS described in Section 7. And as in the monotonic JTMS, in Doyle's JTMS dependencies between nodes are also tracked by means of justifications.

The dependency graph consists of a set N of nodes and a set J of justifications. Justification has the form $< INS, OUTS|c >$ where INS and $OUTS$ are subsets of N: $INS \subseteq N$, $OUTS \subseteq N$ and c is a node: $c \in N$. INS are the monotonic and $OUTS$ the non-monotonic supporters of the conclusion c. A labeling assings each node from N label IN or label OUT. Justification is valid if and only if all its monotonic supporters (INS) are IN and all its non-monotonic supporters ($OUTS$) are OUT. Consistent labeling is a labeling where each IN node has a valid justification. Doyle's JTMS has similar notion of well-foundedness as the monotonic TMSs - cycles in labelings have to be avoided. A labeling is said to be *well-founded* if for every IN node there exists a chaining of valid justifications back to premises or to OUT nodes. A labeling is *admissible* if it is consistent and well-founded. In the dependency network, justifications can create cycles. It is important to watch out for cycles that contain an odd number of OUT nodes, so called odd loops. These cycles express a kind of paradox (liar paradox) and the original Doyle's labeling algorithm was not able to find an admissible labeling in their presence. There is a special node for contradiction designated as $\perp$. Contradiction handling tries to find an admissible labeling in which $\perp$ is OUT.

The original labeling algorithm by Doyle entered an endless loop when it encountered a graph with an odd loop. Goodwin [30, 31] improved this algorithm to at least finish but it still finds no labeling in presence of odd loops. His algorithm is in most cases linear and uses a special ordering of strongly connected components of the dependency graph to guide its computation, for details see Goodwin's papers or [6]. Junker [35] improved the labeling algorithm to always find an admissible labeling whenever one exists but his algorithm is not incremental. Dung proposed a new representation of JTMS and developed an algorithm [22, 23] for it which is incremental and always finds an admissible labeling whenever one exists. Beckstein [6] reports also another complete algorithm created by Russinoff [53]. The complete algorithms are however not practical because the labeling problem is NP-complete [25] so no efficient algorithm is to be expected.

Non-monotonic JTMS was the first one invented - in 1979 by Jon Doyle in his master thesis at MIT. People described many problems of this original system (like being single-context, cumbersome because of non-monotonicity, being described only informally, it was not clear what the relation to non-monotonic logics is) and the work on them resulted in other systems like the ATMS or the LTMS. The most important idea seemed to be the separation between a problem solver and the truth maintenance system which was adopted by all the subsequent TMSs.

11.1 Dual representation

In the previous text, classical representation of the original JTMS, that straightforwardly captures dependencies between nodes, was described. In this section the new representation of JTMS by Dung [22, 23] is presented. This new representation is equivalent to the old one and

makes dependencies between justifications explicit. It makes it possible to separate the monotonic and non-monotonic parts of the dependency graph. Dung also developed an incremental algorithm to compute admissible labelings which uses this new representation.

To describe the new representation and to show its relation to the old one several definitions have to be introduced. Let (N, J) be the classical representation, let K be a subset of J, $K \subseteq J$, then $K \upharpoonright c$ will denote the set of justifications from K which have c as a consequent: $K \upharpoonright c \equiv \{j \in K | j \equiv< INS, OUTS | c >\}$. The new representation translates justifications to nodes but not all nodes need to have a justification in the classical representation. Hence the term completed classical representation: *completed classical representation* is a pair $(N \bigcup NOUT \bigcup \{\Psi\}, J \bigcup JOUT \bigcup \{j_\perp\})$. $JOUT = \bigcup \{j_c\}, NOUT = \bigcup \{t_c\}$ where for every node $c \in N \bigcup \{\perp\}$ which is not a consequent of some justification $(J \upharpoonright c = \varnothing)$ a new node t_c and a new justification $j_c =< \{t_c\}, \{\} | c >$ are added. Let K be a subset of $J \bigcup JOUT$. A *monotonic dependency* $MD(j)$ of the new representation has the form $MD(j) =< In(P(K)) | j >$, where $In()$ denotes in-components and $j \in J \bigcup \{\perp\}$ is called the consequent of $MD(j)$. Similarly, a *non-monotonic dependency* has the form $NMD(j) =< Out(P(K)) | j >$ where Out denotes out-components of the new representation and $j \in J$ is called the consequent of $NMD(j)$.

Let (N, J) be a classical representation of a JTMS and $(N \bigcup NOUT \bigcup \{\Psi\}, J \bigcup JOUT \bigcup \{j_\perp\})$. Then for every $j \equiv< INS, OUTS | c >\in J \bigcup \{j_\perp\}$:

The monotonic dependency is determined as follows:

$$MD(j) = \begin{cases} \{\} & \text{if } INS = \varnothing \\ \{(J \bigcup JOUT) \upharpoonright i_1, \ldots, (J \bigcup JOUT) \upharpoonright i_k\} & \text{if } INS = \{i_1, \ldots, i_k\} \end{cases}$$

The non-monotonic dependency $NMD(j)$ is determined as follows:

$$NMD(j) = \begin{cases} \{\} & \text{if } OUTS = \varnothing \\ \{(J \bigcup JOUT) \upharpoonright o_1, \ldots, (J \bigcup JOUT) \upharpoonright o_m\} & \text{if } OUTS = \{o_1, \ldots, o_m\} \end{cases}$$

Then $(J \bigcup JOUT \bigcup \{j_\perp\}, D)$ is called the *dual representation* of this JTMS where $D = \bigcup_{j \in J \bigcup \{j_\perp\}} (MD(j) \bigcup NMD(j))$.

Notions such as dual labeling, valid dependency, well-founded and admissible labeling are defined for the new representation similarly as for the classical representation. A *dual labeling* is a function from $J \bigcup JOUT \bigcup \{j_\perp\}$ to the set $\{IN, OUT\}$. Monotonic dependency is valid iff each its IN-component contains at least one justification labeled IN and non-monotonic dependency is valid iff all its OUT-components are labeled OUT. Consistent dual labeling is a labeling where each IN justification has valid dependencies. A dual labeling is said to be well-founded iff for every IN justification there exists a chaining of valid dependencies going back to premise justifications, or to OUT justifications. A labeling is *admissible* if it is consistent and well-founded.

Except giving these definitions, Dung also proves that the classical and new representation are equivalent in a sense. More precisely, each admissible labeling of a completed classical representation corresponds exactly to an admissible dual labeling of the new representation and vice-versa. He also develops an algorithm to move from one admissible dual labeling to another which is based on a decomposition of the network in dual representation. DDB must be called only if all admissible dual labellings are inconsistent. The algorithm is essentially a chronological backtracking algorithm for the incremental computation of an admissible and

consistent dual labeling. It uses the observation that in the dual representation the set of non-monotonic dependencies of each justification contains at most one non-monotonic dependency. This facilitates treating the non-monotonic cases.

Dung reports [22] that the transformation between classical and dual representation can be performed in $O(|J| * |N|)$ time. This result together with Elkan's proof of NP-completeness of the problem of finding an admissible labeling [25] leads to the conclusion that the complexity of Dung's algorithm is exponential. Dung also points out that his algorithm has also the advantage of having provisions for using a preference order between assumption justifications (justifications having non-empty OUT-components) which gives the problem solver a greater control over the dual labeling process.

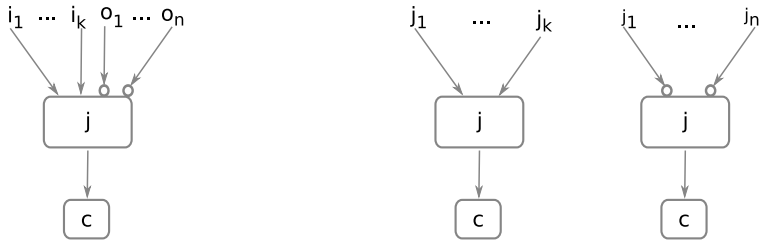

Figure 15: Graphical representation of JTMS dependency graph in the classical representation (the first graph) and the dual representation (the two graphs on the right side).

Example transformation To clarify the transformation between classical and dual representations, an example follows. See Figure 15. The first part of the picture represents a part of the dependency graph in the classical representation: justification $j \equiv < \{i_1 \dots i_k\}, \{o_1 \dots o_n\} | c >$ which connects the assumptions and premises $i_1, \dots, i_k, o_1, \dots, o_n$ to the consequent node c. The second and third part of the picture depict a monotonic dependency $MD(c) \equiv < In(\{j_1, \dots, j_k\} | c >$ and a non-monotonic dependency $NMD(c) \equiv < Out(\{j_1, \dots, j_n\}) | c >$ in the new representation.

Now consider the example in Figure 16. It depicts a part of the Scenario 1 from Section 2.1 as a JTMS dependency graph in the completed classical representation. It is the completed classical representation because the auxiliary nodes $NOUT = \{t_f, t_c, t_r\}$ and justifications $JOUT = \{j_f, j_c, j_r\}$ have already been added. The set of justifications J contains three justifications: $J = \{j_{fs}, j_{cs}, j_{sr}\}$. Monotonic and non-monotonic dependencies of the dual representation are computed as follows:

$MD(j_{fs}) = \{(J \bigcup JOUT) \upharpoonright featureSpec\} = \{\{j_f\}\}$
$MD(j_{cs}) = \{(J \bigcup JOUT) \upharpoonright componentSpec\} = \{\{j_c\}\}$
$MD(j_{sr}) = \{(J \bigcup JOUT) \upharpoonright specification\} = \{\{j_{fs}, j_{cs}\}\}$
$NMD(j_{fs}) = \{\}$ (because $OUTS = \varnothing$ in j_{fs})
$NMD(j_{cs}) = \{\}$ (because $OUTS = \varnothing$ in j_{cs})
$NMD(j_{sr}) = \{(J \bigcup JOUT) \upharpoonright reviewed\} = \{\{j_r\}\}$

These dependencies are represented by the dual dependency network in Figure 17. The dual representation clearly separates the monotonic dependencies $MD(j_{fs}), MD(j_{cs}), MD(j_{sr})$ and the non-monotonic dependency $NMD(j_{sr})$.

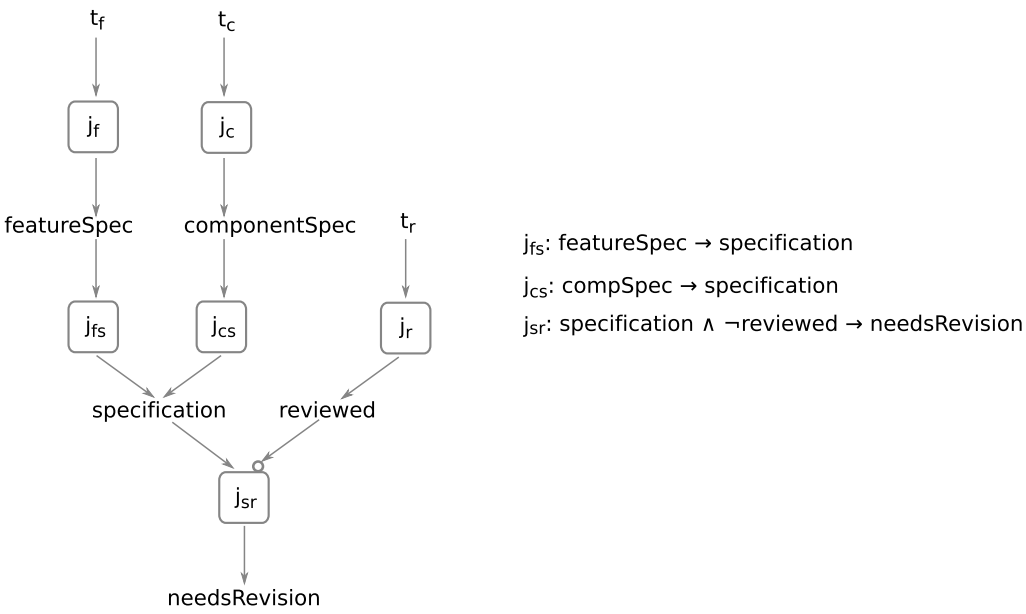

Figure 16: Completed classical representation of part of the Scenario 1 - Specification review.

NMJTMS and non-monotonic logic. For a long time non-monotonic truth maintenance has coexisted with non-monotonic logic while it has been suggested that there is close relationship between them [47] but it was not established precisely. Reinfrank in his paper about logical foundations of non-monotonic truth maintenance [52] sheds more light on this problem. He shows relation of non-monotonic truth maintenance to Konolinge's version of Autoepistemic Logic (AEL) by relating a justification to an AEL-formula of the form $La \wedge \neg Lb \rightarrow c$, where L is a modal operator. Lp is interpreted as "p is believed". Different versions of AEL-extensions then correspond to different restrictions on TMS-labellings. He shows that consistent labellings correspond to weakly grounded AEL-extensions, minimal consistent labellings to moderately grounded AEL-extensions and well-founded consistent labellings to strongly grounded AEL-extensions. An AEL-extension of a theory S is a consistent and self-supporting set of beliefs that can reasonably be entertained on the basis of S [3]. Groundedness refers to the level of self-supportedness of beliefs. It can be said that truth maintenance performs inference in AEL if IN label is identified with L and OUT label with $\neg L$. Establishing relationship between TMSs and non-monotonic logic should allow a logical analysis of non-existence of any labelling or the difference between negated assumptions and OUT-assumptions. For details refer to the Reinfrank's paper [52].

12 Relation to OWL and RDF

This section explores applications of the techniques from previous sections in the area of semantic technologies.

Broekstra and Kampman [8] explore the possibility of using a JTMS-inspired algorithm to remove statements from RDF triple stores. Their algorithm is based on tracking dependencies

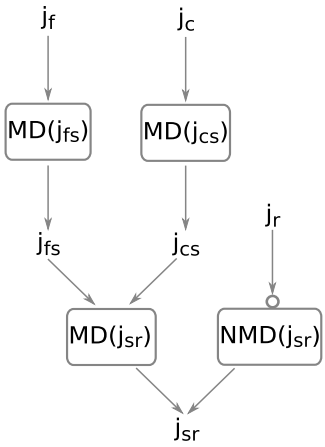

Figure 17: Dual representation to the classical representation in Figure 16.

between triples using justifications. They use justifications to be able to determine consequences of a removed triple that have to be removed as well. They point out that, besides triple removal, the dependency relations between statements are necessary for high-level services on RDF repositories such as change tracking and statement-level security. They implemented the algorithm in the Sesame RDF store. One drawback of their approach is that they had to separate RDF model theory inferences, which are optimized in Sesame to take advantage of RDF model theory rule dependencies, and the algorithm that creates the justification dependency graph. This was necessary because the delete operation needs to know about all possible dependencies of a statement. It is also important to note that Broekstra and Kampman delete all justifications of a statement together with the statement; their approach has a rather superficial resemblance to the TMS algorithms presented in this paper. They conclude that their TMS approach performs satisfactorily for medium sized (roughly up to 200 000 statements) data sets and the expressiveness of RDF and RDF Schema. For comparison, Sesame system itself can cope with data sets of 3 million statements.

Kiryakov and Ognyanov in their paper about tracking changes in RDF(S) repositories [39] also discuss the Sesame TMS subsystem (which is now part of the Sesame project). They make an interesting point that for some applications it is important to be able to distinguish between statements that were explicitly added to the repository and those that deductively follow from the explicitly added statements. Their approach forbids users to remove an inferred statement directly. Removal of an inferred statement then can only be done by removing all of its supporters.

Volz, Staab, and Motik in their paper about incremental maintenance of materialized ontologies [57, 58, 59] mention reason maintenance too but only to express their doubts about its suitability for such a problem. Their doubts concern the fact that in the original TMS algorithms justifications are never removed, only disabled. Their own approach to maintenance of materialized ontologies is based on maintenance of dynamic Datalog programs and implemented in the KAON Datalog engine http://kaon.semanticweb.org/.

The article by Kang and Lau [36] explores the use of belief revision for ontology updates. They motivate their approach by giving a shopping example where two agents communicate

about features of a digital camera. The shop owner and customer each have their own ontology of products which are in conflict with one another though. To resolve that conflict, Kang and Lay use the belief revision expansion, revision and contraction operators (see section 4). They seem to be mixing belief revision with the Dependency Directed Backtracking algorithm used in TMSs but they unfortunately do not explain the details. They refine their approach in a later paper [37] where they do not mention DDB at all and integrate belief revision into an ontological re-engineering method proposed by Gómez-Pérez et al. (2004). The paper concludes by stressing the need to investigate efficient computational methods to be able to apply the technique to large and multiple ontologies.

Gutierrez et al. [33, 34] take a more formal and complete approach to the problem of updates in RDF than the previously presented articles. Their approach uses the Katsuno-Mendelzon framework [38] to define model-theoretic semantics of "erasure operation" on RDF (called this way to distinguish it from the AGM contraction operation). The Katsuno-Mendelzon framework builds on the AGM belief revision theory. Katsuno and Mendelzon argued [38] that no rationality postulates for making belief revision will be adequate for every application and they also clearly distinguished two types of revisions: proper revision and update. The proper revision deals with new information about static world whereas update brings the knowledge base up to date when the world it describes changes. Katsuno and Mendelzon claim that the AGM postulates (see Section 4) describe only revision. Gutierrez et al. concentrate on update which, as they and Katsuno and Mendelzon claim, is prevalent; it includes the updates for which databases are usually used. The approach by Gutierrez et al. is more complete because they consider removing inferred triples too (that is triples that were not added explicitly). They define a semantics for update in RDF and concentrate on the characterization of the erase operation and its consequences over the formulas expressible in RDF. They also define the erasure operator (which is a Katsuno-Mendelzon analog to the contraction operator of AGM belief revision) and show that it cannot be faithfully represented in RDF because RDF has neither negation or disjunction. They show how to approximate the erasure operation by computing erasure candidates which then amounts to finding certain minimal cuts in the RDF graph. The problem of finding whether a graph is an erasure candidate can be done in $PTIME$, as they show. Other recent papers, such as [29], take a similar approach, also based on the Katsuno-Mendelzon framework.

13 Conclusion

Presently, the Semantic web shifts from a read-only web to a read/write web [7] and, at the same time, many traditional web applications turn to social media. These changes bring new requirements on software and information processing in general. The techniques presented in this paper are among the possible tools to build on so as to address the new challenges resulting from the above mentioned changes. The techniques presented in this paper can be seen as complementary forms of so called "defeasible reasoning". This paper focuses primarily on two of them: "belief revision" and "reason maintenance".

Both belief revision and reason maintenance solve a similar problem – how to update existing information. Belief revision takes a more general approach. It studies the properties of rational changes whereas reason maintenance is more pragmatic and focuses on problem solving and updates of knowledge bases. In other words, belief revision tries to capture the nature of beliefs and how they change when confronted with new information whereas reason maintenance serves

mainly as a service to an inference engine to make it easier to build a generic problem solver. This observation is supported by the focus of the papers that have been published in these two fields. Belief revision makes (supposedly) rational changes but encounters problems for example when it has to deal with consecutive changes, i.e., iterated belief revision. Change operators in belief revision depend on epistemic entrenchment which is, however, local to a "belief set" and it is not known what the epistemic entrenchment should look like after an operator is applied; in other words, AGM theory studies one-step revision. This restriction is addressed by so called "iterated belief revision" (see for example [50]). However, according to the latest results [51], iterated belief revision is in conflict with a desired property called *relevance* as formalized by Parikh. Relevance is defined as follows: When a belief state φ is revised by new information ψ, only the part of φ that is related to ψ should be affected; the rest should remain the same. Belief revision papers seem to be striving for an ideal, rational update process. They consider applications only "afterwards" (with the exception, perhaps, of multi-agent systems). On the other hand, reason maintenance papers tend to focus on practical issues: more efficient problem solving in case of Doyle's JTMS, more efficient model-based diagnostic software in case of de Kleer's ATMS and HTMS, embedded real-time execution kernel for which ITMS was used. Reason maintenance approaches do not try to always make rational changes. They sometimes even make a random, or arbitrary choice when it has to be chosen which assumption to remove so as to restore consistency. An attempt, based on the so called cognitive dissonance theory, has been made to replace these random, or arbitrary choices by a more human-like choice [54].

Both belief revision and reason maintenance have a large body of scientific literature, particularly belief revision is a very lively field, so a more focused research would be needed to determine the feasibility of these approaches in the area of social media. On the other hand, with a practical application in mind, more seems to be in favour of reason maintenance due to its more pragmatic grounding.

In both cases, however, extensions and adaptations of these approaches would be necessary to meet the needs of social media that were partially identified in the motivating scenarios (Sections 2.1 and 2.2) and in [9]. These needs are: paraconsistency, explanation, and coping with rules with variables, i.e., beyond propositional logic and the Boolean case traditionally considered in reason maintenance and beleif revision.

Most of the theories presented in this paper are based on consistency and usually are within propositional logic, i.e., they do not admit rules with variables. As seen from the Specification review and Paraconsistency scenarios given in Section 2, social software would benefit from the stronger expressiveness offered by a richer language allowing for variables – it is desirable to be able to express for example a rule like R_4: $needsRevision(x) \wedge reviewer(x, r) \rightarrow todo(x, r)$, whih states that everything needing a revision and that has been assigned to a reviewer is on the reviewer's todo list. The Specification review scenario (Section 2.1) assumes a work in progress towards some goal. If a tool is to support a work in progress, then it has to cope with inconsistencies, as inconsistencies in general cannot be avoided in work in progress. Hence the requirement for paraconsistency. Neither reason maintenance nor belief revision support paraconsistent reasoning. Reason maintenance algorithms typically remove inconsistency as soon as it occurs and the main stream of belief revision is based on deductive closures of belief sets, i.e., implicitly assumes that belief is consistent. Semantically enhanced social software are likely to have a rather complex logic and even more so if it supports non-monotonic reasoning and paraconsistency. Therefore it is desirable to help users by providing explanations. Providing explanations is a difficult issue and, in reason maintenance, it is usually limited to delivering the justification dependency tree. We sketched a similar, basic approach to a simple form of

explanation in [9]. There is no such a clear way to produce explanations in belief revision that we are aware of.

References

[1] Carlos E. Alchourron, Peter Gardenfors, and David Makinson. On the logic of theory change: Contraction functions and their associated revision functions. *Theoria*, 48:14–37, 1982.

[2] Carlos E. Alchourron, Peter Gardenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *The Journal of Symbolic Logic*, 50:510–530, 1985.

[3] G. Aldo Antonelli. Non-monotonic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2006.

[4] Grigoris Antoniou and Mary-Anne Williams. *Non-Monotonic Reasoning*. The MIT Press, 1997.

[5] Krzysztof R. Apt. Some remarks on boolean constraint propagation. *New Trends in Constraints*, pages 91–107, 2000.

[6] Clemens Beckstein. *Teubner-texte zur Informatik - Band 18 - Begründungsverwaltung*. B. G. Teubner Verlagsgesellschaft, 1996. (in German).

[7] T Berners-Lee, J. Hollenbach, Kanghao Lu, J. Presbrey, E. Pru d'ommeaux, and mc schraefel. Tabulator redux: Writing into the semantic web. Technical report, School of Electronics and Computer Science - University of Southampton, http://eprints.ecs.soton.ac.uk/14773/, 2007.

[8] Jeen Broekstra and Arjohn Kampman. Inferencing and truth maintenance in rdf schema - exploring a naive practical approach. *Workshop on Practical and Scalable Semantic Systems (PSSS)*, 2003.

[9] François Bry and Jakub Kotowski. Towards reasoning and explanations for social tagging. *Proc. of ExaCt2008 - ECAI2008 Workshop on Explanation-aware Computing. Patras, Greece*, http://www.pms.ifi.lmu.de/publikationen#PMS-FB-2008-2, 2008.

[10] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[11] Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28, 1986.

[12] Johan de Kleer. Problem solving with the atms. *Artificial Intelligence*, 28:28, 1986.

[13] Johan de Kleer. Exploiting locality in a tms. *Proceedings of AAAI-90*, pages 264–271, 1990.

[14] Johan de Kleer. A hybrid truth maintenance system. Technical report, Xerox Palo Alto Research Center, 1994.

[15] Johan de Kleer and Greg Harris. Truth maintenance systems in problem solving. Technical report, Xerox PARC, 1979.

[16] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[17] Johan de Kleer and Brian C. Williams. Diagnosis with behavioral modes. *Readings in model-based diagnosis*, pages 124–130, 1992.

[18] Simon Dixon and Norman Foo. Beyond foundational reasoning. In *5th Australian Joint Conf. on Artificial Intelligence, World Scientific*, 1992.

[19] Jon Doyle. Truth maintenance systems for problem solving. Technical Report AI-TR-419, Department of Electrical Engineering and Computer Science of MIT, 1978.

[20] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[21] Jon Doyle. The ins and outs of reason maintenance. *IJCAI'83*, 1983.

[22] Truong Quoc Dung. A new representation of jtms. *LNCS*, 990/1995, 1995.

[23] Truong Quoc Dung. A revision of dependency-directed backtracking for jtms. *LNCS*, 1137, 1996.

[24] Thomas Eiter and Georg Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Proc. of the 11th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 261–273, 1992.

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

[26] John O . Everett and Kenneth D . Forbus. Scaling up logic-based truth maintenance systems via fact garbage collection. *Proc. AAAI-96*, pages 614–620, 1996.

[27] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers*. MIT Press, 1993.

[28] Peter Gärdenfors, editor. *Belief revision*. Cambridge University Press, 1992.

[29] Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the approximation of instance level update and erasure in description logics. *Proc. of the National Conference on Artificial Intelligence*, 22(1):403, 2007.

[30] James W. Goodwin. An improved algorithm for non-monotonic dependency net update. Technical report, Linkoping University, Department of Computer and Information Science, 1982.

[31] James W. Goodwin. *A Theory and System for Non-Monotonic Reasoning*. PhD thesis, Linkoping University, 1987.

[32] Nikos Gorogiannis and Mark D. Ryan. Implementation of belief change operators using bdds. *Studia Logica*, 70:131–156, 2002.

[33] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Updating semantic web data. page 11, 2005.

[34] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. The meaning of erasing in rdf under the katsuno mendelzon approach. *Proceedings WebDB-2006*, page 6, 2006.

[35] Ulrich Junker. Variations on backtracking for tms. *Proc. of the 1990 Workshop on Truth Maintenance Systems, ECAI 90*, 1990.

[36] Seung Hwan Kang and Sim Kim Lau. Ontology revision using the concept of belief revision. *Proceedings of the 8 thInternational Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES-04), Springer*, 2004.

[37] S.H. Kang and S.K. Lau. Ontology revision on the semantic web: Integration of belief revision theory. *Proc. of the 40th Annual Hawaii International Conference on System Sciences*, 2007.

[38] Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Belief Revision*, pages 387–394. Cambridge University Press, 1991.

[39] Atanas Kiryakov and Damyan Ognyanov. Tracking changes in rdf(s) repositories. *Transformation for the Semantic Web KTSW 2002*, 2002.

[40] Robert Koons. Defeasible reasoning. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2005.

[41] Paolo Liberatore and Marco Schaerf. Reducing belief revision to circumscription (and vice versa). *Artificial Intelligence*, 93:261–296, 1997.

[42] Joao Pavao Martins. *Reasoning in multiple belief spaces*. PhD thesis, State University of New York at Buffalo, 1983.

[43] David A. McAllester. A three valued truth maintenance system. *AI-Memorandum rept., MIT*, 1978.

[44] David A. McAllester. An outlook on truth maintenance. 1980.

[45] David A. McAllester. Truth maintenance. *AAAI90*, 1990.

[46] D. McDermott. Contexts and data dependencies: A synthesis. *IEEE Trans. Pattern Anal. Mach. Intell.*, 3:237–246, 1983.

[47] Drew McDermott and Jon Doyle. Non-monotonic logic i. *MIT A.I. Memo 486*, 1978.

[48] P. Pandurang Nayak and Brian C. Williams. Fast context switching in real-time propositional reasoning. *Proc. 14th Nat. Conf. AI*, 1997.

[49] Bernhard Nebel. Belief revision and default reasoning: Syntax-based approaches. In *Principles of knowledge representation and reasoning*, pages 417–428. Morgan Kaufmann, 1991.

[50] Pavlos Peppas. *Handbook of Knowledge Representation*, chapter 8 - Belief Revision, page 42. Elsevier, 2007.

[51] Pavlos Peppas, Anastasios Michael Fotinopoulos, and Stella Seremetaki. Conflicts between relevance-sensitive and iterated belief revision. In *18th European Conference on Artificial Intelligence, ECAI2008, Patras, Greece*, pages 85–88, 2008.

[52] Michael Reinfrank. Logical foundations of nonmonotonic truth maintenance. *Proc. EPIA 89, Fourth Portuguese Conference on Artificial Intelligence, Springer*, 1989.

[53] D. Russinoff. An algorithm for truth maintenance. Technical report, MCC, 1985.

[54] Peter J. Schwartz. Truth maintenance with cognitive dissonance. Technical report, University of Mariland at College Park, 2001.

[55] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence 9*, pages 135–196, 1977.

[56] Mladen Stanojevic, Sanja Vranes, and Dusan Velasevic. Using truth maintenance systems - a tutorial. *IEEE*, page 11, 1994.

[57] Raphael Volz, Steffen Staab, and Boris Motik. Incremental maintenance of dynamic datalog programs - extended abstract.

[58] Raphael Volz, Steffen Staab, and Boris Motik. Incremental maintenance of materialized ontologies. *Proc. of ODBase, Springer*, 2003.

[59] Raphael Volz, Steffen Staab, and Boris Motik. Incrementally maintaining materializations of ontologies stored in logic databases. *Data Semantics II-LCNS*, 3360:1–34, 2004.

[60] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. *Proceedings of AAAI-96*, 1996.

[61] C. Williams. Art the advanced reasoning tool: Conceptual overview. *Inference Corporation, Los Angeles*, 1984.

[62] Marianne Southall Winslett and Marianne Winslett. *Updating Logical Databases*. Cambridge University Press, 1990.

[63] Guofu Wu and George Macleod Coghill. A propositional root antecedent itms. *Proceedings of the 15th International Workshop on Principles of Diagnosis*, 2004.