INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

An Incremental Approach to Knowledge Assimilation

Vanessa Klaas

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Beginn der Arbeit:	15.10.2003
Abgabe der Arbeit:	15.09.2004
Betreuer:	Prof. Dr. François Bry
	Dr. Norbert E. Fuchs, Universität Zürich

Abstract

A controlled language like Attempto Controlled English (ACE) together with knowledge assimilation is an useful tool for many applications, e.g. writing correct specifications. In practice, knowledge assimilation must be implemented efficiently because the amount of data can grow very large. The incremental approach to model generation described in this report reduces computations by memorizing the results already computed. Further searches use these results by continuing at the end of the last search and not by restarting at the beginning. This proceeding obviously avoids computing same things repeatedly.

Contents

1 Introduction

1.1 Motivation: Attempto Controlled English and Knowledge Assimilation

The motivation of this work is Attempto Controlled English (ACE). ACE is a research project of Dr. Norbert Fuchs at the Department of Informatics of the University of Zurich, Switzerland.

A controlled language is a natural language which is restricted in grammar and vocabulary to disambiguate texts written in such a language. This ensures both that readers who are familiar with the controlled language, do not missunderstand texts written in the controlled language and that such texts are easy to automatically process. So, controlled languages are powerfull tools for writing precise texts like industrial specifications. ACE is a controlled English, i.e. a subset of (natural) English. Due to its non-ambiguousness, ACE can be translated into first-order logic which allows reasoning on texts written in ACE: texts can be queried, inconsistencies and redundancies detected.

An objective of current research related to ACE is knowledge assimilation. The task of knowledge assimilation is to update an existing knowledge base (a finite set of formulas), i.e. to add new formulas to a knowledge base or to remove formulas from a knowledge base. There are many different approaches to knowledge assimilation because there is a big variety of use cases for knowledge assimilation with totally different situations and conditions (e.g. available time and space for computation) where knowledge assimilation is applied: an agent acting and reasoning in an environment or an author editing a text in ACE. In this work, the latter case is considered.

As sketched above, ACE is a restriction of natural English. By controlling the language, one disambiguates texts while retaining their readability: in contrast to a formal specification language, a text written in ACE is as readable as a text in (natural) English. Of course, the reader must be aware of the controlling aspect of ACE and has to know its grammar. There are situations where missunderstandings can have serious consequences, e.g. a mechanician might missunderstand the specification of maintenaince of an airplane's component. On the other hand, ACE is a useful tool for engineers who need to specify a complex matter. By respecting the grammar of ACE, the process of text editing results in a precise specification almost automatically and texts are written in a clear style. Since ACE is a subset of natural English, it is easier to learn than a formal language. One does not need being an expert in English to be able to learn ACE. The non-ambiguousness of ACE leads to another advantage, namely ACE is translatable unambigously into first-order logic. Therefore, ACE can be processed by computers. While ACE texts are well readable for humans, they can be input for computer programs at the same time. So, ACE is applicable everywhere where precise input is needed.

To reach this profitable non-ambiguousness, some aspects of natural English must be controlled. As described by Uta Schwertel in [Sch00], a simple sentence where plural is occuring can be interpreted in several different ways and all of them are correct in natural English. Therefore, plural is controlled by defining its semantics explicitly. Another problem of natural language is the interpretation of anaphores. ACE solves it by specifying a rule how to treat anaphores. Passive constructs are not allowed to ensure that each sentence has a subject. The grammar of ACE is given by Stefan Hoefler in [Hoe04].

Besides some basic vocabulary, one can use a lexical editor to define domain specific vocabulary. One has to specify explicitly the grammatical properties of the words that shall be used.

In spite of these restrictions, ACE is a powerful language that did not loose too much of expressiveness of full language and moreover, is translatable unambiguously into first-order logic as mentioned above. This is done in two steps. An input consisting of ACE sentences is translated into discourse representation structures which are already a form of first-order logic, but representing the logical statements in a different way. In a second step, the usual representation in form of clauses is generated.

Therefore, ACE can be input of a theorem prover or model generator to solve reasoning tasks. The existing ACE reasoner is called RACE and bases on SATCHMO, a model generator implemented in Prolog. Norbert Fuchs gives some details in [FS02].

There is already an Attempto system implemented which consists of a lexical editor and the reasoner RACE among other things. The latter one can also be used for knowledge assimilation.

In general, the task of knowledge assimilation is to integrate new input into an existing knowledge base. In the context of ACE, the knowledge base is represented by a text written in ACE, new input consists of new ACE sentences.

Assume the task of writing a specification in ACE. Such a specification does not appear from nowhere, the process of writing a text takes time, even a completed specification may change: if the specified object is further developed, the specification must be adapted or, the other way round, a product specification is further specified before the development of the specified product advances. Therefore, besides new sentences which are added to an existing text, also deleting sentences must be considered. Furthermore, some sentences may simply change, but this case can be reduced to the other two operations: changing a sentence can be seen as deleting the old version and then adding the new version of it.

An important and desirable property of a text, especially of a specification, is consistency. A technical specification containing a contradiction cannot be correct and would be hardly useful. Therefore, after having edited a text, a check of (in-)consistency is useful. Another desirable property is to have a text free of redundancies. But there might be special cases where redundancy might be whished. So, a powerful tool for text editing might be an interactive system which warns the user if it detects redundancy and it is the user's task to decide how to treat it. On one side, redundancy can reduce the difficulty of a text (by repeating complex aspects in different ways), on the other hand, short texts can be also the goal. Anyway, each time, a single sentence of a text is changed, knowledge assimilation should happen. Since even a singleton ACE sentence in general results in a set (containing more than one element) of first-order logic formulas, one assumes that insertions or deletions do not refer to single formulas but to sets of formulas.

Now it is helpful to recall the relationship between the changes of a text, the old text and the new one, i.e. consistent or not, redundant or not – or simply independent. In case of redundancy, there are two possibilities: either, the new input (or a part of it) can be derived from the already existing text or a part of the existing text now can be derived from the other part together with the new input. This relationship can be formalized as Robert Kowalski has done in [Kow94]: the relationship between a given state T of a theory (i.e. knowledge base), an input sentence P and a successor state T' of the theory can be classified in four different cases:

1. P is a logical consequence of T, i.e. $T \models P$.

2. Part T_1 of T is logically implied by P together with the other part T_2 of T, i.e. $T = T_1 \cup T_2, T_1 \cap T_2 = \emptyset$ and $\{P\} \cup T_1 \models T_2$.

3. P is inconsistent with T, i.e. $\{P\} \cup T \models \perp$.

4. None of the relationships mentioned above holds.

To realize knowledge assimilation in the Attempto system, two aspects must be considered. The first one is how to detect *efficiently* which case of the four mentioned above applies and secondly, how to proceed in presence of an inconsistency or a redundancy in a knowledge base.

1.2 Goal of This Work

As described above, reasoning on ACE texts is already implemented in the Attempto system. The reasoner is called RACE and uses the model generator Satchmo. There is a naive approach implemented: if one would like to test a text regarding inconsistency, Satchmo is started once with the input consisting of the logical formulas representing the text. A model (i.e. a set of ground atoms) found by Satchmo means that the text is consistent. If now one would like to check the text for redundancies, Satchmo is executed a second time. A text A follows logically from a text T, i.e. A is redundant in T, if Satchmo started with the input consisting of the logical formulas representing T together with the negated logical formulas representing A. If this computation results in a contradiction, then A is redundant.
This proceeding is a little bit naive because things are computed twice instead of making use of the results of the first computation. It would be more efficient to "simply" remember all that was computed by Satchmo the first time and use it to solve the second task.

For the second aspect mentioned above, i.e. the treatment of new input, one needs efficient methods of adding or deleting formulas to, from resp., the knowledge base (represented as a finite set of logical formulas). An efficient method for detecting the right case of input also is needed, but the detection of the applying case can be reduced to adding formulas. If a model of the union of the "old" set and of the "new" set is found, the new input is not inconsistent. Now add the negated formulas of the "new" set to the "old" one and search a model. If Satchmo finds a model, then the new input is not redundant (of the same tenor as the first case formulated by Kowalski). To test for the second sort of redundancy, one needs to test for each formula F of the "old" set, if the "old" set without F together with the negation of F and the new input leads to a contradiction. In this case, F is a logical consequence of the other formulas. The whole procedure is shown in section 4 more precise.

Due to the functioning of Satchmo, namely generating a tree with branches that represent interpretations, the approach presented in this work is *incremental* model generation. This means that a model found for a text together with the whole tree expanded up to now is memorized and, like the process of editing is, *incrementally* adapted to changes of the text. "Incrementally" means to stop expanding the search tree as soon as a model is found (for a consistent set of formulas) and, for a changed set of formulas, to restart searching for a probably new model not at the root, but at the last model found, i.e. continue expanding the memorized tree.

2 Preliminaries

2.1 Some Basic Notations

In the following chapters, some basic notations are used.

Definition 1 (polyary junctors). *The binary junctors, namely disjunction and conjunction, must be extended to n-ary equivalents, where n is a natural number.*

- $\wedge^\star(F_1, \dots, F_n)$ *denotes the conjunction of n formulas* $F_1, \dots, F_n$*. For* $n = 0$*,* $\wedge^\star()$ *denotes "true". It is also called "top" and written as* $\top$*.*
- $\vee^\star(F_1, \dots, F_n)$ *denotes the disjunction of n formulas* $F_1, \dots, F_n$*. For* $n = 0$*,* $\vee^\star()$ *denotes "false", also called "bottom" and written as* $\bot$*.*

In this work, the formulas considered have a special form which is also introduced in [BY96]. They are in *implication form* and range-restricted.

A clause in implication form looks like

$$\wedge^\star(A_1, \dots, A_n) \Rightarrow \vee^\star(B_1, \dots, B_m),$$

where $A_1, \dots, A_n, B_1, \dots, B_m$ are atoms. $\wedge^\star(A_1, \dots, A_n)$ is called *body* and $\vee^\star(B_1, \dots, B_m)$ is called *head* of the clause. Such a clause is called *range-restricted* if all variables occuring in the head are also present in the body.

Another notation used later concerns relations. Let S be a set and R be a relation on S, i.e. $R \subset S \times S$. Then $\overline{R}$ means the *reflexive-transitive closure* of R.

2.2 PUHR Tableaux

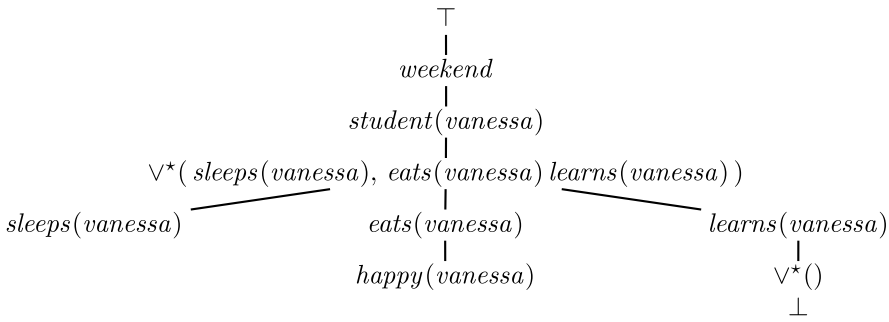
Let S be a finite set of range-restricted clauses in implication form. PUHR tableaux are used to generate models for S or to demonstrate that there does not exist anyone. A PUHR tableau is a tree whose nodes consist of atoms or disjunctions of atoms. It is constructed by application of two different rules. By the first one, a node can be added if its atom is a conclusion of the atoms of the branch of the node together with the clauses of S. By the second rule, a disjunction can be splitted, which means that a node containing a disjunction consisting of n atoms gets n successor nodes, one for each atom of the disjunction.

Branches of PUHR tableaux can be closed which means that they represent interpretations that are not models of S. If all branches of a PUHR tableau are closed than the tableau also is called closed and there does not exist any model, i.e. there exists a formula $F \in S$ that is inconsistent to $S \setminus \{F\}$.

The following examples illustrate this.

Example 1. *For the following set, a PUHR tableau is given.*

$$\begin{aligned}
C_1 &= \wedge^\star() && \Rightarrow \vee^\star(weekend)\\
C_2 &= \wedge^\star() && \Rightarrow \vee^\star(student(vanessa))\\
C_3 &= \wedge^\star(student(vanessa)) && \Rightarrow \vee^\star(sleeps(vanessa),\ eats(vanessa)\ learns(vanessa))\\
C_4 &= \wedge^\star(student(vanessa),\ weekend) && \Rightarrow \vee^\star(sleeps(vanessa),\ eats(vanessa))\\
C_5 &= \wedge^\star(learns(vanessa)) && \Rightarrow \vee^\star()\\
C_6 &= \wedge^\star(eats(vanessa)) && \Rightarrow \vee^\star(happy(vanessa))
\end{aligned}$$

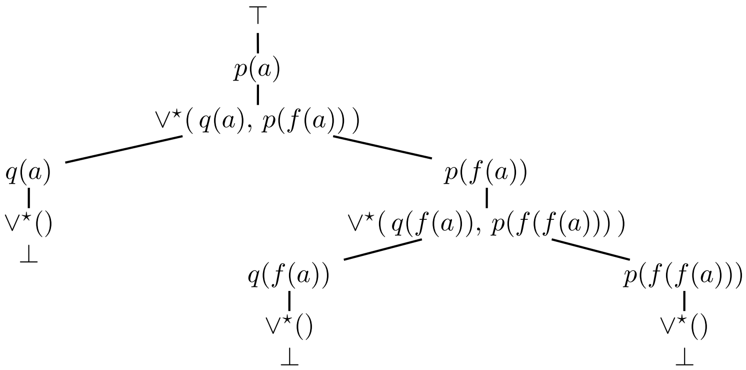


Models are:

- $\{weekend, student(vanessa), sleeps(vanessa)\}$
- $\{weekend, student(vanessa), eats(vanessa), happy(vanessa)\}$
- *The rightmost branch of the tableau is closed and therefore not a model.*

Example 2. *The following set is inconsistent, i.e. there does not exist any model: the PUHR tableau is closed (i.e. every branch of it is closed).*

$$\begin{array}{ll}
\wedge^\star() & \Rightarrow \vee^\star(\, p(a)\,) \\
\wedge^\star(\, p(x)\,) & \Rightarrow \vee^\star(\, q(x),\ p(f(x))\,) \\
\wedge^\star(\, p(x),\ q(x)\,) & \Rightarrow \vee^\star() \\
\wedge^\star(\, p(f(x)),\ p(f(f(x)))\,) & \Rightarrow \vee^\star()
\end{array}$$



The formal definition of PUHR tableaux is the following, originally given in [BY96].

Definition 2. (PUHR tableaux) *Positive unit hyperresolution (PUHR) tableaux for a set $\mathcal{S}$ of clauses in implication form are (finite or infinite) trees whose nodes are sets of ground atoms and disjunctions of ground atoms. Finite PUHR tableaux for $\mathcal{S}$ are inductively defined as follows:*

1. *$\{\top\}$ is a positive unit hyperresolution tableau for $\mathcal{S}$.*
2. *If T is a positive unit hyperresolution tableau for $\mathcal{S}$, if L is a leaf of T such that an application of the PUHR rule (resp. splitting rule) to formulas in L yields a formula E (resp. two formulas E_1 and E_2) not subsumed by an atom in L, then the tree T' obtained from T by adding the node $L \cup \{E\}$ (resp. the two nodes $L \cup \{E_1\}$ and $L \cup \{E_2\}$) as successor(s) to L is a positive unit hyperresolution tableau for $\mathcal{S}$.*

Infinite PUHR tableaux for $\mathcal{S}$ are defined as follows: If $(T_i)_{i\in\mathbb{N}}$ is an infinite sequence of finite PUHR tableaux for $\mathcal{S}$ such that for all $i \in \mathbb{N}$ T_{i+1} results from an application of a PUHR tableau expansion rule to T_i, then $T = \bigcup_{i\in\mathbb{N}} T_i$ – i.e. the tree T with Vertices$(T) = \bigcup_{i\in\mathbb{N}}$ Vertices(T_i) and Edges$(T) = \bigcup_{i\in\mathbb{N}}$ Edges(T_i) – is a PUHR tableau for $\mathcal{S}$.

A branch of a positive unit hyperresolution tableau is said to be closed, if it includes a node containing the atom $\perp$. A positive unit hyperresolution tableau is said to be closed if all its branches are closed. A branch (resp. tableau) which is not closed is said to be open.

A positive unit hyperresolution tableau T for $\mathcal{S}$ is said to be satisfiable if the union of $\mathcal{S}$ with the nodes of a branch of T is satisfiable.

Remark. *Note that this definition makes no difference between a node and its content (i.e. a set of formulas). In section 3, we have to leave this elegant definition because a more complex structure for PUHR nodes is needed.*

3 Applying PUHR Tableaux to Knowledge Assimilation

As described in section 1, knowledge assimilation in the context of ACE can be realized in different ways. The simple method is using an existing model generator which is executed each time one needs to check for (in-)consistency or redundancy of new input. In doing so, same things are computed for several times.

The approach presented now tries to void this. PUHR tableaux are applied to generate a model. Everything computed while searching a model and the model itself is memorized. The result in form of a partially expanded tableau is used for further computations. For another computation, the existing tableau is just adapted to the updated knowledge base and model generation is continued starting from a leaf of the adapted PUHR tableau.

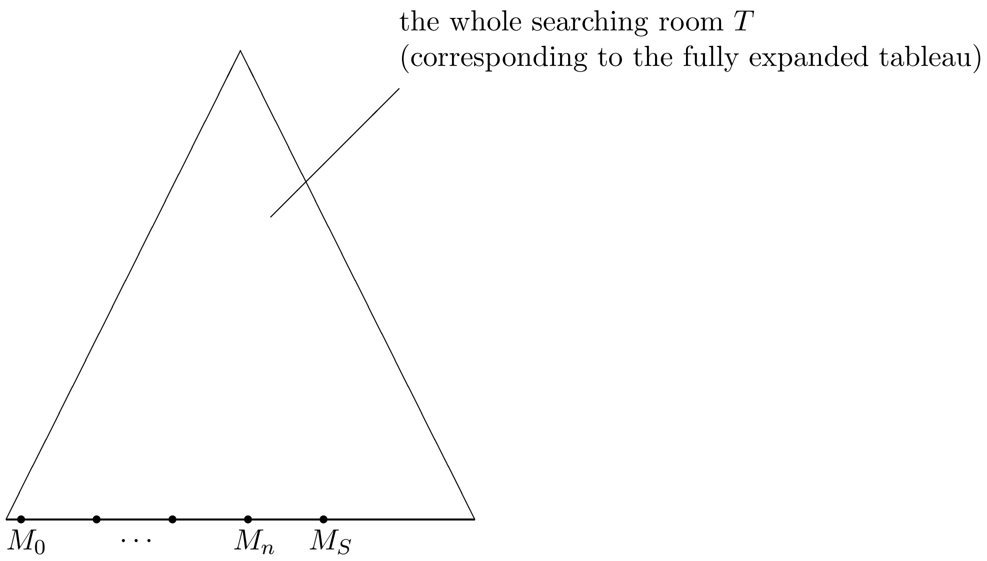
3.1 Approach to Incremental Model Generation

For a given finite, consistent set S of formulas, we have found a model M_S by model generation with PUHR tablaux. Before we have found M_S, we probably have generated some branches (i.e. interpretations of S) $M_0, \dots, M_n$ that are not models for S.
Remember the whole tree generated up to now inclusive the model M_S. When adding a finite, consistent set A of formulas to S, take the existing model M_S and search for a model of $S \cup A$ *starting from* M_S. Figure 3.1 illustrates this idea.
When deleting a formula $F \in S$, the tableau has to be adapted, i.e. some branches may disappear and some interpretation discarded previously may be a model of the new set $S \setminus F$.
The advantage of this approach is that every branch is computed only once. Especially the case of adding formulas demonstrates this.

3.2 Extended Ordered PUHR Tableaux

Definition 2 does not restrict the choice of the next node for expansion. If there is a node N in a tableau that has two children N_1 and N_2, it does not make any difference if one expands N_1 first and then N_2 or the other way round, for the only important thing is the end result: is there any model and how does it look like? For the purposes of knowledge assimilation as sketched in section 1, we need more – namely a sort of ordered expansion such that always the leftmost-depth-first expansible leaf is expanded – because we would like

Figure 1: main idea

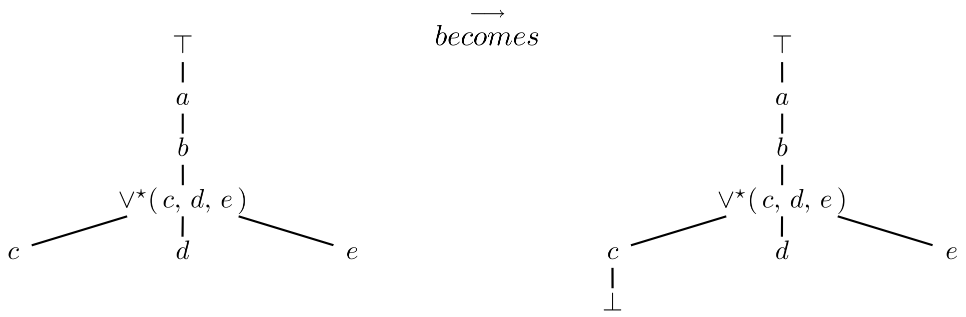


to record partially expanded tableaux. Thus, the original definition must be extended by a leftmost-depth-first order. One could take any other total order, but the leftmost-depth-first order is the one which is already implemented in Satchmo. Therefore, this is an extension of the mathematical definition which is already implemented.
For adding formulas to a consistent, finite set S, these ordered PUHR tableaux would be sufficient, whereas the deletion of formulas is more complicated: some branches may disappear. Moreover, closed branches can be reopened, which is a special case of disappearing branches. Therefore, when deleting a formula, we need to check if one $M_i, 0 \leq i \leq n$, of the search room in figure 3.1 is a model of $S \setminus \{F\}$. Thus, the definition of PUHR tableaux must be extended by an additional structure which allows to trace how a node was generated, i.e. each node and each formula in S has an unique identifier, a set $\mathcal{R}$ of identifiers of the "reasons" that contains the clauses needed to construct the node.

To ease the definition of extended ordered PUHR tableaux, I give some examples first – the first one is to illustrate the total order, the second one shows the extended node structure.

Example 3. *To keep the example simple, consider a set of variable-free clauses (in implication form). Under the left column, the appropriate partially expanded tableau (only the formulas of the left column were used) and the tree order is given.*
The tableau on the right hand side arises from the one on the left hand side by applying the PUHR rule to the smallest expansible leaf (c) with use of the last formula of the right column.

$\wedge^\star() \Rightarrow \vee^\star(a)$	$\wedge^\star(b, c) \Rightarrow \vee^\star()$
$\wedge^\star() \Rightarrow \vee^\star(b)$	$\wedge^\star(a, e) \Rightarrow \vee^\star(f)$
$\wedge^\star(a, b) \Rightarrow \vee^\star(c, d, e)$	

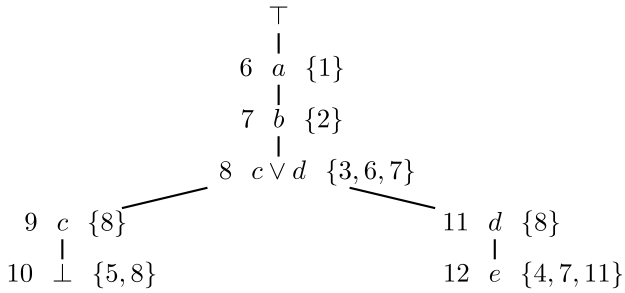


$a \leq b \leq \vee^{\star}(\, c,\, d,\, e\,) \leq c \leq d \leq e$

Notice that in contrast to "ordinary" PUHR tableaux, each node consists only of the single formula written in the graphical tree, and not of the set of the formulas appearing in the branch from $\top$ to the node. In this example, the identifiers and the set of reasons which every node contains, are omitted. This structure is illustrated in the following example:

Example 4. *For the following set, an extended ordered PUHR tableau is given. The identifier is written on the left of each node, the set of "Reasons" on the righthand side of each node.*

1	$\wedge^{\star}()$	$\Rightarrow$	$\vee^{\star}(\, a\,)$
2	$\wedge^{\star}()$	$\Rightarrow$	$\vee^{\star}(\, b\,)$
3	$\wedge^{\star}(\, a,\, b\,)$	$\Rightarrow$	$\vee^{\star}(\, c,\, d\,)$
4	$\wedge^{\star}(\, b,\, d\,)$	$\Rightarrow$	$\vee^{\star}(\, e\,)$
5	$\wedge^{\star}(\, c\,)$	$\Rightarrow$	$\vee^{\star}()$



In the preceding example, one could argue that the node closing the left branch should get the identifier 11 instead of the node d with identifier 11 by the formal definition. But renumbering results in the example above. Just change the siblings on the right-hand side containing atoms of the splitted disjunction. Another possibility would be to use rational numbers for having ne nodes numbered in the same order than the leftmost-depth-first order. So, a general definition is given now.

Definition 3 (extended ordered PUHR tableaux). Extended leftmost-depth-first ordered PUHR tableaux *for a set S of clauses in implication form are (finite or infinite) ordered trees $T = (\mathcal{T}, \leq_{\mathcal{T}})$ with a leftmost-depth-first order $\leq_{\mathcal{T}} \subseteq \mathit{Vertices}(\mathcal{T}) \times \mathit{Vertices}(\mathcal{T})$ on the nodes of $\mathcal{T}$.*

- *Each formula F of S or of a node N of $\mathcal{T}$ has an unique identifier $\mathit{identifierOf}(F) : S \cup \bigcup_{N \in \mathit{Vertices}(\mathcal{T})} \mathit{formulaOf}(N) \to \mathbb{Q}$.*

- *A node N consists of a ground atom or a n-ary disjunction of ground atoms together with its identifier and a set of "reasons" $\mathcal{R}_{identifierOf(F)}$ which contains the identifiers of the formulas (the occurences of the formulas resp.) that has been used to derive the occurence of the formula F.*

Finite PUHR tableaux for S are inductively defined as follows:

1. *$\{\top\}$ is an extended ordered PUHR tableau for S with identifierOf($\top$) $:= c_0$, $\mathcal{R}_{c_0} := \emptyset$, c_0 not an identifier for a formula in S, and it holds: $\top \leq_{\mathcal{T}} V$ for all $V \in$ Vertices($\mathcal{T}$)*

2. *Let L be a leaf of $\mathcal{T}$.*

   (a) *Application of the PUHR rule to formulas in relevantClauses(L) $:= S \cup \bigcup_{N \in ancestors(L)}$ formulaOf(N) yields a formula E not subsumed by an atom in relevantClauses(L), i.e. the PUHR rule is applied to formulas $A_1 \wedge \ldots \wedge A_n \rightarrow E, A_i, 1 \leq i \leq n$, with identifierOf($A_1 \wedge \ldots \wedge A_n \rightarrow E$) $= c_k$ and identifierOf(A_i) $= c_{k_i}$.*
   Then an extended ordered successor tableau $T' = (\mathcal{T}', \leq_{\mathcal{T}'})$ is defined as follows:
   Vertices($\mathcal{T}'$) $:=$ Vertices($\mathcal{T}$) $\cup \{E\}$
   Edges($\mathcal{T}'$) $:=$ Edges($\mathcal{T}$) $\cup \{(L, E)\}$
   identifierOf(E) $:= c_l$ where c_l is a "new identifier", i.e. for all $F \in L \backslash \{E\}$: identifierOf(F) $\neq c_l$ and
   $R_{c_l} := \{c_k, c_{k_1}, \ldots, c_{k_n}\}$.
   $\leq_{\mathcal{T}}$ is extended into $\leq_{\mathcal{T}'} \subset$ Vertices($\mathcal{T}'$) $\times$ Vertices($\mathcal{T}'$) as follows:

   $$\leq_{\mathcal{T}'} := \overline{\leq_{\mathcal{T}} \cup \{(L, E)\}}$$

   (b) *Application of the splitting rule to a formula in L yields n formulas $E_1, E_2, \ldots, E_n$ not subsumed by an atom in relevantClauses(L), i.e. the splitting rule is applied to a formula $\vee^\star(E_1, \ldots, E_n)$ with identifierOf($\vee^\star(E_1, \ldots, E_n)$) $= c_k$.*
   Then an extended ordered successor tableau $T' = (\mathcal{T}', \leq_{\mathcal{T}'})$ is defined as follows:
   Vertices($\mathcal{T}'$) $:=$ Vertices($\mathcal{T}$) $\cup \{E_1, \ldots, E_n\}$
   Edges($\mathcal{T}'$) $:=$ Edges($\mathcal{T}$) $\cup \{(L, E_1), \ldots, (L, E_n)\}$
   identifierOf(E_j) $:= c_{l_j}$ where $c_{l_j}, 1 \leq j \leq n$, are pairwise different "new identifiers", i.e. for all j and for all $F \in L \backslash \{E_j\}$: identifierOf(F) $\neq c_{l_j}$ and identifierOf(E_i) $\neq$ identifierOf(E_j) for all $i \neq j$
   $R_{c_{l_j}} := \{c_k\}$ for all j.
   $\leq_{\mathcal{T}}$ is extended into $\leq_{\mathcal{T}'} \subseteq$ Vertices($\mathcal{T}'$) $\times$ Vertices($\mathcal{T}'$) as follows:
      i. *$\leq_{\mathcal{T}'} := \overline{\leq_{\mathcal{T}} \cup \{(L, E_1) | \; 1 \leq i \leq n\} \cup \{(E_i, E_j) | \; 1 \leq i < j \leq n\}}$ (because E_1 is "left" of E_2 is "left" of $\ldots E_n$ in the formula $\vee^\star(E_1, \ldots, E_n)$)*
      ii. *for all $i < j$ and for all $V \in$ successors(E_i) : $V \leq_{\mathcal{T}'} L \cup \{E_j\}$.*

Notice that the order defined above is a total order on the nodes of an extended ordered PUHR tableau.

Definition 4 (ordered expansion). *Let S be a consistent, finite set. Let $T = (\mathcal{T}, \leq_{\mathcal{T}})$ be a partially expanded ordered PUHR tableau for S. $T' = (\mathcal{T}', \leq_{\mathcal{T}'})$ is a* (direct) successor (ordered) PUHR tableau of T *if and only if* T *is obtained by applying the PUHR rule, splitting rule resp., as defined in definition* 3*, to some formula, formulas resp., in relevantClauses*(L)*, where* L *is the smallest expansible leaf.*

Remark. *Notice that because of the order* $\leq_{\mathcal{T}}$*, the choice of the* node *is now deterministic, but the choice of the* formula, formulas *resp., of a set* S *on which a rule should be applied, rests nondeterministic. This is an important aspect allowing us to add new formulas without changing the "old" tableau. It is part of the implementation to propose a method to choose well. One possible strategy is to apply the PUHR rule as often as possible before applying the splitting rule to minimize the number of branches.*

3.3 Adding Formulas and X Tableaux

Initially, a consistent, finite set S of clauses with an appropriate PUHR tableau T_S is given. Now, a consistent, finite set A of "new" formulas should be added to S. This assimilation consists of two steps:

1. Adding the set A to S
2. Actualize the appropriate PUHR tableau, i.e. continue expanding the already existing tableau beginning at the leaf of the branch that represents the existing model of S

After the first step, the branch representing the model M_S of S is not fully expanded for $S \cup A$ in general, i.e. M_S possibly neither represents a model of $A \cup S$ nor is M_S closed. For the lack of a better name and to have a short denomination for such kind of extended ordered PUHR tableaux, call it *X tableau.* An X tableau for a set of formulas is not closed and there does not exist a branch of it that represents a model for the given set of formulas.

Definition 5 (X tableaux). *Let* S *be a finite set of formulas with* $S \not\models \perp$*. Let* $T = (\mathcal{T}, \leq_{\mathcal{T}})$ *be an extended ordered PUHR tableau. Let* A *be a finite set of formulas with* $A \not\models \perp$*.*
$\delta_A(S) := S \cup A$ *and for each* $F \in A$: *identifierOf*$(F) := c_F$*, where* c_F *is a "new" identifier, i.e. for all* $N \in$ *Vertices*$(\mathcal{T}), G \in S \cup A \setminus \{F\}$: *identifierOf*$(F) \neq$ *identifierOf*(N) *and identifierOf*$(F) \neq$ *identifierOf*(G)*.*
Now define $\delta_A(T) := T$*. This notation should demonstrate that* T *is seen as a PUHR tableau for* $\delta_A(S)$*. It is called* X tableau *for* $\delta_A(S)$*, i.e.* $\delta_A(T)$ *is not closed and does not specify (necessarily) a model for* $\delta_A(S)$*.*

Proposition 6. *Let* S, A *be finite, consistent sets of formulas,* $T = (\mathcal{T}, \leq_{\mathcal{T}})$ *a PUHR tableau of* S*.*
Then all closed branches B *in* T *are also closed in* $\delta_A(T)$*.*

Proof. Let B be a closed branch in T, M a leaf of B. Then holds $M \not\models S$, i.e. there is $F \in S : M \not\models F$. Since $F \in S$, also $F \in S \cup A$. Therefore holds $M \not\models S \cup A$. □

Remark. *S may be extended by A, because the sequence of formulas chosen from S is unordered. Belated adding of formulas seems as if the formulas of A had been there right from the beginning, but not choosen until now.*
The only point where some changes may occur is the model M of S. Since the set S of formulas has been extended, M probably is no longer a model of S. Therefore, the extended tableau is not necessarily a PUHR tableau representing a model*, so it is called* X tableau*. By resuming the ordered expansion of this X tableau, it becomes a PUHR tableau* representing a model *again as shown in the next proposition.*

Definition 7. *Let S, A be finite, consistent sets of formulas, let $T = (\mathcal{T}, \leq_{\mathcal{T}})$ be a PUHR tableau of S.*
Then $T_{+A} = (\mathcal{T}_{+A}, \leq_{\mathcal{T}_{+A}})$ is an X tableau, which arises by ordered expansion of the X tableau. $\delta_A(T)$.

Proposition 8. *T_{+A} is an extended ordered PUHR tableau (specifying a model if not closed) of $S \cup A$.*

Proof. Let S, A be finite, consistent sets of formulas, $T = (\mathcal{T}, \leq_{\mathcal{T}})$ be an extended ordered PUHR tableau of S specifying a model $M \models S$.

Case 1 $M \models S \cup A$. Then $T_{+A} = \delta_A(T)$ is an extended ordered PUHR tableau for $S \cup A$ with model M.

Case 2 $M \not\models S \cup A$. Then $M \not\models A$. So, there exists some $F \in A : M \not\models F$. Continue with ordered expansion for (all such) F.

□

3.4 Deleting Formulas

The initial situation is like above: we have a finite, consistent set S of clauses in implication form. In addition, we have a partially expanded ordered PUHR tableau with n closed branches $M_0 \leq \ldots \leq M_{n-1}$ and a model $M_{n-1} \leq M_n = M_S$ of S. Now, a single clause $F \in S$ should be deleted.

There are three things to do when deleting a formula F:

1. Adjust the tableau, i.e. remove the nodes whose generation depends directly or indirectly on the formula F. Notice that some branches that were closed before can be reopened by the deletion of F.

2. Remove F from S.

3. Find the leftmost model of $S \setminus \{F\}$, i.e. test for the remaining branches $M_i, i \leq n$, if anyone is a modell of $S \setminus \{F\}$.

3.4.1 Algorithm for Deletion

Let S be a finite, consistent set of formulas, $T_S = (\mathcal{T}_S, \leq_{\mathcal{T}_S})$ an extended ordered PUHR tableau for S, M_n a model of S. Let $F \in S$ be the formula which should be deleted and $c_F :=$ identifierOf(F). Let D be the set of nodes which are marked for deletion. In the first step, D must be constructed.

- Starting at the root $\top$ depth-first searching on T_S, check for every node N, $c_N :=$ identifierOf(N):
  If $c_F \in$ reasonsOf(N) or there exists some $c \in D$ with $c \in$ reasonsOf(N).
  Then update D: $D := D \cup \{c_N\}$.

- Delete all nodes N with identifierOf(N) $\in D$. The result is an X tableau $\overline{T} = (\overline{\mathcal{T}}, \leq_{\overline{\mathcal{T}}})$. In this step, one special case must be considered: a node N should be deleted that is root of a subtree containing at least two branches which are not completely marked for deletion. The simplest way to treat this is to keep the leftmost branch and to discard the others.

- Delete F, i.e. $S_{-F} := S \cup \{F\}$.

Remark. *The tableau is searched from top to bottom, i.e. from the root to the leaves, because first of all, it is quite natural since PUHR tableaux are defined like that. Moreover, this searching order keeps the algorithm simple:*
it needs to traverse the tableau only once. This can be seen in the following way: Let D be the set of marked nodes. Suppose, D is extended by some c_i while checking a node N_i. Then the algorithm does not need to restart the search with this new extended set D because for all nodes $N \leq_{\mathcal{T}} N_i$ it holds: $c_i \notin$ reasonsOf(N), i.e. all nodes N that are "before" N_i, has been generated at a moment at which N_i did not exist, therefore, N_i could not have been used to generate N and c_i cannot be in their sets of reasons.
This argumentation does not hold if the algorithm starts search at the leaves!

3.4.2 Algorithm for Finding a (Possibly) new Model

Test for all remaining branches (from the "left side" to the "right side" $M_i, 0 \leq i \leq n$ if $M_i \models S_{-F}$.

- If there exists a number $i \leq n$ with $M_i \models S_{-F}$, then a model $M_{S_{-F}}$ is found and nothing else must be done and $T_{S_{-F}} := \overline{T}$.
- If $M_i \not\models S_{-F}$ for all $i \leq n$, then $T_{S_{-F}}$ is the resulting X tableau by ordered expansion of $\overline{T}$.

Proposition 9. *The X tableau $T_{S_{-F}}$ is an ordered extended PUHR tableau for S_{-F}.*

Proof. As in proposition 8. □

4 Further Considerations

With the tools of chapter 3, it's possible to realize knowledge assimilation: one is able to add formulas to or remove formulas from a knowledge base S and having always an associated PUHR tableau and a model. In this chapter, a proposal is given, how "the four cases of input" mentioned in section 1 may be recognized and treated.
In this chapter, assume a finite, consistent set S of formulas and a second one, named $A = \{F_1, \dots, F_n\}$, which should be added to the first one.

4.1 Detecting Inconcistencies

The detection of an inconsistency of $S \cup A$ is now trivial: if A is inconsistent together with S, there does not exist any interpretation of $S \cup A$ that fulfils the whole set $S \cup A$. The consequence is that none of the interpretations is a model and all branches of the PUHR tableau for $S \cup A$ are closed. Therefore, an inconsistency is found if the extended ordered PUHR tableau resulting by ordered expansion for S_{+A} is closed.
In presence of an inconsistency, one can nondeterministically search the subset $A_\perp \subset A$ of A containing at least one formula F that causes the inconsistency or inconsistencies, i.e. for each formula $F \in A_\perp$ it holds $S \cup \{F\} \models \perp$, which means that $S \cup \{\neg F\}$ is consistent. Therefore, by using the method of adding formulas, search for a model of $S \cup \{\neg F\}$. If a model is found,

F belongs to $A_\perp$.
For realization, the tableau for S must not be discarded, for one has to restart the tableau generation several times beginning at the tableau associated to S.

4.2 Detecting Redundancies

As already mentioned in section 1, there exist two sorts of redundancy. The simple case is that the new input in form of a set A of formulas already follows logically from the knowledge base S. The other case is that one part of S follows logically from the other part of S together with the new input A. Both sorts of redundancies are detected in the same way: execute model generation for a modified set S (this is also done for testing regards to consistency). In both cases, negated formulas are used and not finding any model by incremental model generation means the presence of redundancies.

The first sort of redundancy is detected as follows:
Let $\neg A$ be the set of the negated formulas of A, i.e. $\neg A := \{\neg F_1, \ldots, \neg F_n\}$ for $A := \{F_1, \ldots, F_n\}$.
Now try to find a model for $S \cup \neg A$, i.e. add $\neg A$ to S as described in section 3. If incremental model generation has not found any model, i.e. adding $\neg A$ to S leads to a contradiction, there exist redundancies. To find them, one needs to add $\neg A$ step by step, i.e. only one formula at once.

The detection of the second sort of redundancy is a little bit more complicated:
One has to test for all formulas $F \in S$ if $(S \setminus \{F\}) \cup \neg A \cup \{\neg F\}$ leads to a contradiction. If a contradiction is found, then F is redundant. This can be done by deleting F from S and adding $\neg A \cup \{\neg F\}$ to S as described in section 3.

Like in the case of inconsistency, the original tableau for S must be memorized during the whole procedure because incremental model generation is executed several times. Furthermore, only the presence, absence resp., of redundancies is detected. After that, the tableau of S is needed for updating S in the desired way appropriate to the gained informations up to now.

4.3 Treatment of new Input

The question which arises now is, how to treat a detected inconsistency in form of a subset $A_\perp$ and how does the way of treating detected redundancies looks like? This question is not part of this work because the answer depends on the situation in which knowledge assimilation is applied. For every situation there are several options: the task is to decide which formulas should be kept, e.g. the older or the newer ones. Perhaps, one would like to have a minimal set which must be discarded.

In the case of an author editing a text in ACE, the question depends on the intention of the author and the purposes of the text. Indeed it is possible that some redundancies are whished (in contrast to contradictions), e.g. for a better understanding of the text. So, in this situation the question could be solved as follows: an user of an interactive editing system writes a text. The he executes a sort of validation. The system reports inconsistencies if any present. After having eliminated them, the user revalidates and the system reports the redundancies found in the text. For each redundant ACE sentence, the user has the possibility to keep or to eliminate it (further revalidations are needed because by deleting one redundant, inconsistent resp., sentence, another one perhaps is not redundant, inconsistent resp., any more).

5 Conclusion

By the previous sections a solid mathematical basis for knowledge assimilation is established. One is able to add formulas to a knowledge base and to remove formulas from a knowledge base. All the time, there exists a PUHR tableau that is also updated every time the knowledge base is updated. This guarantees that after each update, a model is specified for the current (consistent) knowledge base. This together with the (partially) expanded PUHR tableau is used to generate the next model incrementally, i.e. one does not need to recompute the already existing part of the tableau.

An implementation is missing yet. It should be done in Prolog because the original Satchmo also implemented in Prolog shows that Prolog is a good choice.

Besides the implementation, this work can be starting point for other interesting tasks like a Satchmo program that does not operate on first-order logic clauses, but on discourse representation structures (DRS) which are the intermediate structure occuring during the translation of ACE sentences into first-order logic formulas. If one had a sort of "DRS"-Satchmo, the second translation can be avoided which possibly results in a gain of efficiency.

References

[BY96] François Bry and Adnan Yahya. Minimal model generation with positive unit hyper-resolution tableaux. In Pierangelo Miglioli, Ugo Moscato, Daniele Mundici, and Mario Ornaghi, editors, *Theorem Proving with Tableaux and Related Methods, 5th International Workshop, TABLEAUX'96, Terrasini, Palermo, Italy*, volume 1071, pages 143–159. Springer-Verlag, 1996.

[FS02] Norbert E. Fuchs and Uta Schwertel. Reasoning in attempto controlled english. Technical report, IFI, University of Zurich, July 2002.

[Hoe04] Stefan Hoefler. The Syntax of Attempto Controlled English: An Abstract Grammar for ACE 4.0. Technical Report ifi-2004.03, Department of Informatics, University of Zurich, Zurich, Switzerland, 2004.

[Kow94] R. Kowalski. Logic without Model Theory. In D. M. Gabbay, editor, *What is a Logical System?*, pages 35–71. Oxford University Press, 1994.

[Sch00] Uta Schwertel. Controlling plural ambiguities in Attempto Controlled English. In *Proceedings of the 3rd Internaltional Workshop on Controlled Language Applications*, Seattle, Washington, 2000.