Robert Eschbach
Inger Sonntag

PLATIN
A Planning System for Inductive Theorem Proving
Implementation and Experiences

Fachbereich Informatik, Universität Kaiserslautern, Postfach 3049, D-67653 Kaiserslautern,
{eschbach,sonntag}@informatik.uni-kl.de, SEKI-Report SR-97-02, March 24, 1997

Abstract

This paper provides a description of PLATIN. With PLATIN we present an implemented system for planning inductive theorem proofs in equational theories that are based on rewrite methods. We provide a survey of the underlying architecture of PLATIN and then concentrate on details and experiences of the current implementation.

Contents

1 Introduction **1**

2 Architecture **2**

3 Sensors **5**
3.1 Sensor Structure 5
3.2 Generation of Sensors 6

4 Methods **8**
4.1 Method Structure 8
4.2 Generation of Methods 10

5 Planning Tree **12**
5.1 Node Structure 12
5.2 Representation 14
5.3 Propagation 14

6 Conflict Manager **16**
6.1 Main Procedure 16
6.2 Planning Strategies 17
6.3 Heuristic Choose-Node 19
6.4 Heuristic Refinement-or-Verification 21

7 Verifier **24**
7.1 Abstract Verifier 24
7.2 Verifier Method 25
7.3 UNICOM 26

8 Example **27**
8.1 Planning Process 27
8.2 Conjectures 34
8.3 User Interaction 34

9 Conclusions **37**
9.1 Review 37
9.2 Future Lines 38

A User Interface **39**

B Example **41**
B.1 Sensor and Method Descriptions 41
B.2 Specification qsort 43
B.3 Verifier Output 48

C Functions **53**

References **56**

Index **59**

1 Introduction

In this paper we will present a highly interactive system called PLATIN for planning inductive theorem proofs in equational theories that are based on rewrite-methods. It should be noted that although PLATIN is restricted to inductive theorem proofs, the underlying concept allows theorem proving, in general. This restriction has practical reasons as it allowed us to relatively quickly implement a system in order to test the feasibility of our concept and to evaluate its functionality for a range of examples. For more information concerning inductive theorem proofs in equational theories refer to [Ba88], [Gr89] and [Av95]. PLATIN has been implemented on SPARC-Stations 10 using (common) Lisp and Garnet, a user interface development environment for Lisp (cf. [My et al.91]).

This paper is organized as follows: In section 2 an overview of the *architecture* and the principles underlying PLATIN will be given. It has also been presented in [SoDe93] and will therefore only be covered briefly. In sections 3,4 5, 6 and 7 the basic components of PLATIN - namely sensors, methods, planning tree, conflict manager and verifier - will be described. The structure of *sensors* and *methods* as well as how they are created will be explained. To specify a *planning tree* we will provide a description of the nodes' structure and will illustrate the so-called propagation, an essential feature ensuring the consistency of the planning tree at every stage of the planning process. The *conflict manager*, which can be seen as the control unit of PLATIN, will be characterized by its main procedure, its planning strategies and two heuristics concerning the search of a 'working node' and the decision on refinement or verification. Every *verifier* integrated in PLATIN has to meet some correctness and functionality requirements. These and the general verifier method, which specifies the cooperation between a verifier and the other components of PLATIN, will be explained. Thereafter we will briefly outline the current verifier employed by PLATIN. In section 8 a somewhat larger *example* will be presented and section 9 will provide some *concluding remarks* concerning this paper. The *appendix* is divided into three parts. In appendix A offers information concerning the activation and use of PLATIN's user-interface. Appendix B technical details of the example described in section 8 can be found. The function employed in the examples throughout the paper are listed in appendix C.

2 Architecture

Let us now take a closer look at the general ideas underlying PLATIN. We believe planning to be one of two interleaving steps of a proof process, the other being verification of a theorem. We characterize planning as a deduction of problems (later referred to by hypothesis) from the original one, such that they presumably enable a verification or rejection of the latter [1]. In the case of PLATIN new equations are not only derived from the current equation to be proved but also from information provided by the system itself. The derived equations are viewed as lemmata by the verifier, the verification component, when the current equation is to be verified. Clearly, their verification or rejection has to be considered, too. As the equations generated by planning can either be wrong or not useful for the verification, it is advisable to interleave these two actions. A failed verification can also reveal information to the planning component as to how the verification can be helped along. Therefore, planning can also be seen as a means to overcome some of the verifier's short-comings. This implies that the input/output behaviour of the verification and the planning component of a proof system have to be adjusted to each other. In addition, one planning component can in principle be devised to work with several different verifiers. PLATIN, however, has up to now only one underlying verifier, called UNICOM. For more detailed information on UNICOM refer to [GrLi91]. As our studies have shown, considerable more theorems can be proved automatically by a proof system consisting of PLATIN and UNICOM than by UNICOM alone.

Considering the above notions, a *plan* for a specific problem can then be regarded as a structure that reflects not only the proposed deductions but their state with respect to verification as well. In PLATIN (see figure 1) we have realized a data-structure referred to as *planning tree*. Each planning tree represents such a plan. Planning trees are described in detail in section 5. At this stage it suffices to know that each node of a planning tree in PLATIN consists of an equation and parameters that further describe the equation and how it is generated from the parent-node. All equations of direct children of a node are considered to influence the verification process of the parent node's equation. The proof process ends successfully if a complete plan has been generated and verified. That is to say that each node of which the equation has been labeled as "used" has to be verified modulo its child-nodes' equations. The equation of a node will be labeled as "used" if it is used in the verification of the parent-node's equation. The proof process then consists of two types of steps, proof steps and verification steps. A verifier run is considered to be a verification step, while a proof step refers to the expansion of the planning tree by one or several nodes that form one derivation.

The construction of a planning tree has to be carefully controlled such that it is not only correct but also leads to a complete plan. For this purpose PLATIN is equipped with a control unit called *conflict manager*, described in section 6. Its name is derived from the kind of decisions that are made by the conflict manager. After each proof step the conflict manager determines what kind of step is to follow, either a verification or a planning step, and to which node of the current planning tree it will be applied. In addition, there usually exists a variety of planning steps that can be applied to a node. This follows from the nature of theorem proving (or problem solving in general) and is the reason we believe planning to be essential for theorem proving. Therefore the necessity of PLATIN to provide a means to formulate planning steps, which we have called *methods* (section 4), naturally follows.

[1] Note, that planning divides a problem into subproblems but its (in-)correctness does not automatically follow from the (in-)correctness of the subproblems

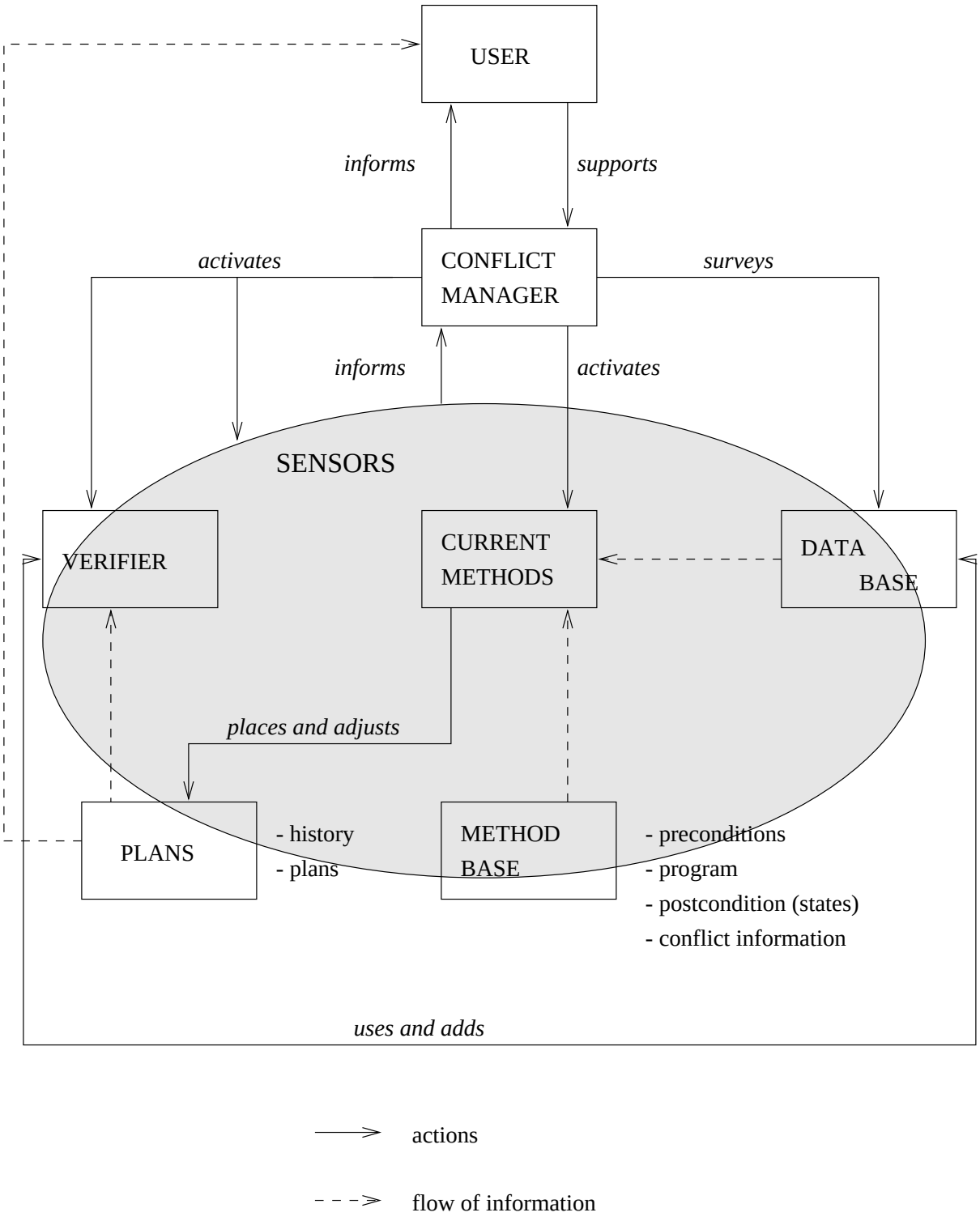

Figure 1: The Architecture Underlying PLATIN

A range of methods has already been added to PLATIN. They are stored in the so-called method base. However, it is possible for the user to devise his own methods and add them to the method base. Experiments have shown that this is required especially when proving examples from new and untried domains. The support of the development of new methods is one reason why PLATIN allows the user to interrupt the automatic proof process (controlled by the conflict manager) in order to switch to an interactive mode of the system. During the latter the user can choose applicable methods, decide on the node to be processed next and so on.

To ensure the conflict manager works correctly and efficiently its decision procedure has to be supplied with appropriate information. To such an end we have introduced the concept of *sensors* (section 3) into PLATIN. Sensors can be seen as small, independent programs that can evaluate all system parameters. The output of sensors is available to the conflict manager at all times and referred to when needed. For example, a sensor can imply a certain structure of an equation; it can be used to judge the history of a node, thus helping to judge its importance or it can be used to evaluate the parameters of the verifier during a verification step. When the conflict manager chooses a planning step as the next proof step and has decided on a node, it then has to decide which methods can be applied. To aid this decision, methods themselves are expected to provide information concerning their applicability. This can be achieved by referring the conflict manager to appropriate sensors by mentioning them in the methods. This service of the methods for the conflict manager often leads to the generation of new sensors when a user generates new methods. Therefore, the base of sensors provided by PLATIN can be extended by the user. In addition to information concerning their applicability, methods provide the conflict manager with information concerning their success. That is to say, they include a rating as to how useful the equations following from their application will be in the verification process of the current node's equation. In general, there is more than one method applicable to a node and the conflict manager then has to choose one (or more). Note, that the rating is not the only parameter to influence this decision.

Sensors are furthermore used to analyse the underlying *data base*, which, in turn, aids the conflict manager's choice of methods. The data base provides the specifications as well as a range of lemmata. For practical reasons, PLATIN's data base has been constructed according to the demands of UNICOM. This implies that the defining rules of the specifications are restricted to terminating, left-linear rules. In addition every non-constructor symbol has to be totally defined and all the defining rule systems have to be locally (ground) confluent. From this also follows the use of free constructors, only.

3 Sensors

The process of planning naturally involves the evaluation of a lot of parameters. PLATIN not only enables an assessment of system parameters but also allows it to be goal-oriented by featuring the concept of sensors. Sensors are predicates on system parameters. Any decidable predicates on system parameters can be combined to define a sensor, returning a truth value for the combination whenever it is required in the planning process. From this also follows that sensors do not change the parameters of the system but only assess them for evaluation. Note, that by system parameters we do not only refer to statistical information about the system but also to information concerning the structure of the current domain, the data-base and so on. For example, a sensor can test whether the syntactical form of the current equation is of a special form or whether a method in question has already been used in the planning process.

The current version of PLATIN does not feature a hierarchical organization of sensors. However, with the application of PLATIN to new domains the number of sensors can possibly rise so drastically that such an organization will be needed for structural reasons and for efficiency. A further enhancement of PLATIN is planned concerning the assistance of the user when defining new sensors. So far sensors are defined with the help of a LISP-Macro as being described below.

3.1 Sensor Structure

A sensor consists of a unique name, a set of sensor-variables, a natural description of its behaviour and a program. The state of a sensor is represented by its variables, the so-called *sensor-variables*. The variables will not be initialized at the moment of activation but retain their old values. This way a sensor can remember old settings, e.g. the number of leafs in the planning tree during the last activation. At the beginning sensor-variables are unbound. Each of these sensor-variables is locally defined in a sensor. This implies that the value of a sensor-variable can only be changed by the sensor itself and different sensors can possess sensor-variables with the same names. To ensure this, sensor-variables can only be changed by special functions, which are given in the following section. Sensor-variables also serve as the interface to the respective sensor. That is to say, that the values of sensor-variables can be used by other parts of the system. For example, methods can refer to sensor-variables in their preconditions, which allow a decision on whether a method is applicable or not. Such a reference consists of the name of the respective sensor followed by the name of a sensor-variable. An activation of a sensor leads to the application of its *program*. The value returned by this program will be referred to as the value of the respective sensor or its main information. *Description* consists of an informal explanation of the sensor's program which should allow a quick interpretation of the program's return values. The structure of sensors is reflected in the following example.

Example 3.1: *Let currentTheoremLhs be the left hand side of the currently analysed equation and fs-sort a build-in function returning the sort of a function-symbol.*

```
NAME                sensor1
SENSOR-VARIABLES    ?sort
DESCRIPTION         sensor1 returns true if currentTheoremLhs is of
                    the following form: the leading function-symbol
                    of currentTheoremLhs is 'ord and currentTheoremLhs
                    has only one subterm that is not a variable and
                    that is of sort LIST
PROGRAM             if currentTheoremLhs = ord(?sort(t))
                       and fs-sort(?sort) = LIST
                    then return(true)
                    else return(false)
```

3.2 Generation of Sensors

In this section we describe how new sensors in PLATIN are created. Basically this will be done in a Lisp environment in which a macro named `defun-sensor` allows the generation of new sensors. In the moment of creation, i.e. evaluation of the defun-sensor expression, the sensor will become part of PLATIN. The components of a sensor, as described above, are: name, variables, description and program. This enumeration coincides with the parameter list of the macro `defun-sensor`. With the help of the example of the previous section, the use of the macro (defun-sensor) in order to create and add a new sensor to PLATIN, is illustrated.

Example 3.2: *Line 1 contains the sensor name 'sensor1' and the list of sensor-variables '(?sort)'. Line 2 represents the description, the remaining lines form the program. In case of activation, the main information of the sensor refers to the value the program returns. In this case it can be either 'true' (line 10) or 'nil' (line 11).*

```
1 (defun-sensor sensor1 (?sort)
2 "the value of ?sort occurs in (lhs current-theorem) "
3    (when (fs-equal (fs-top (lhs (get-current-theorem))) 'ord)
4       (let* ((term (second (subterms (lhs (get-current-theorem)))))
5              (sort (fs-top term)))
6         (cond ((and (not (varp term))
7                     (= (fs-arity sort) 1)
8                     (sort-equal (fs-sort sort) 'list))
9                (set! ?sort sort)
10               'true)
11              (t nil)))))
```

The sensor-variable '?sort' in the example is not an ordinary lisp-variable. For example, it is impossible to set this variable with the lisp-form 'setq'. For this purpose, special functions are provided for sensor-variables, that are listed below. In the example, line 9, the variable '?sort' is changed by the function 'set!'.

`set! <sensor-variable> <new-value>`
sets the value of `<sensor-variable>` to `<new-value>`

`unset! <sensor-variable>`
after this call `<sensor-variable>` is unbound
`bound? <sensor-variable>`
checks, whether `<sensor-variable>` is bound or not
`value <sensor-variable>`
returns the value of `<sensor-variable>`

There are two possibilities to insert a new sensor into PLATIN. Firstly, the respective macro can directly be loaded into the LISP environment. Secondly, it is possible to create a file that only consists of sensor macros. This file can then be included into PLATIN from the user-interface, see appendix A. When PLATIN is loaded, a set of default sensors is provided that can be found in $(PLATINHOME)/sensors/sensors.platin. The user is allowed to adapt this file to his own wishes, adding or deleting sensor macros. However, before deleting a sensor it must be insured that this sensor is not referred to in any other part of the system.

4 Methods

A main feature of the planning process presented in this paper is the deduction of new equations from an equation to be proved. The means to formulate such a planning step are methods. Methods provide a procedure by which new equations are generated. These new equations will also be referred to as decompositions, deductions or derivations. Note, that deductions do not guarantee the verification of the equation they have been derived from. They might not even be logically correct equations. It is therefore important to carefully restrict the use of methods to promising situations and to give a rating of the predicted usefulness of its derivations, depending of the planning state. Such restrictions, also called preconditions, are predicates and as such are able to include sensors. As in the case of sensors, no hierarchical organization of methods has been realized in PLATIN. However, it might be advisable at a later stage to ensure efficiency and a clear structure.

4.1 Method Structure

A method consists of a unique name, a natural description of its behaviour, a slot allowing the type of a method to be set to dynamic or static, a precondition which allows a decision about the applicability of the method, a set of sensors which will be accessed by the precondition, a conflict information which weighs the method in a conflict situation as described in section 6, a postcondition ensuring facts or giving hints about what is achieved by the method and finally a program to be executed. A method is identified by a unique *name* and is described by its *natural description*. However, the type of a method needs to be further explained. There exist two types of methods, static and dynamic ones. A *static method* is a method that, when applied to the same node more than once, would always lead to the same result, the same deductions. For example, a method generating hypotheses from the current theorem by using only the definitions of the predicates and functions used in the theorem as well as the theorem, itself, is a static one. A *dynamic method*, on the other hand, can have differing derivations as results whenever it is applied to the same node. An example for a dynamic method is one that generalizes the current theorem. Depending on the form of the current theorem there can exist several distinctive generalizations of the same kind, each of which presents one decomposition of the current theorem. It follows, that a static method only needs to be applied once to a node while a dynamic one can be applied as often as it results in a new derivation. To ensure that dynamic methods are appropriately applied, the notion of a notebook has been added to the contents of a node of a planning tree, see section 5. A node's notebook allows methods to put down remarks concerning the respective node, only. This implies that dynamic methods must comprise elements to analyse the notebook's contents, appropriately, and to add notes to it. If an application of a dynamic method to a node does not lead to new results, its name will be added to the list of used methods of this node, called forbidden-methods. All methods that are referred to in this list will not be applied again to the respective node. It follows that static methods are added to after just one application.

Let us now consider a situation in which more than one method can be applied to a node. In section 6 this is referred to as a conflict situation as the Conflict Manager has to solve the conflict of which method to activate next. To avoid a random decision the conflict manager must be supplied with information concerning the worth of an application of a method at the time. Some parts of the information are provided by sensors, others by the methods itself. To this end, every method contains a *conflict information*, a function which has the planning tree as its argument and returns a natural number. The evaluated conflict

information is also called the weight of the respective method. Presently, PLATIN provides the following types of conflict information [2].

1. planning `<node>`
   1 if the node is a leaf, 0 otherwise

2. replanning `<node>`
   2 if the node is no leaf and has one rejected son, 0 otherwise

3. refinement `<node>`
   1 if the node is no leaf and has no rejected sons, 0 otherwise

However, the user is also expected to form the conflict information of new methods according to his perception of the worth of the method for the planning process. Further information to be used by the conflict manager is provided by postconditions. In contrast to the conflict information the postcondition refers to the usefulness of the equations created by the respective method in regard to the verification process. For example, if a method generally leads to deductions that are easily verifiable, this can be expressed in the postcondition. This parameter is then added to the new nodes as additional information, see also 5. The conflict manager can take this into account when deciding whether the next step for the respective node is to be a planning step or a verifier step. Presently, PLATIN features three types of postconditions:

1. maybe-verifiable `<node>`

2. insufficient `<node>`

3. verifiable `<node>`

If the derivations resulting from a method are generally not sufficient to allow the verification of the parent-node, this should be noted by the method in form of the postcondition `insufficient(newNode)`. Note, that `newNode` refers to the deductions induced by the method, while `currentNode` refers to the node the method is applied to (the parent node of the new deductions). The postcondition `maybe-verifiable(newNode)` reflects that the deductions will probably be verifiable without further planning steps. The postcondition `verifiable(currentNode)`, on the other hand, expresses that the deductions have been constructed such that the equation of their parent node is without doubt verifiable (cf. section 7.1). Methods guaranteeing this postcondition will be called 'strong methods'. The latter would, for example, be the case, if the derivations were obtained by generalization. In which way the conflict manager makes use of the information provided by the postcondition is explained in section 6. The structure of a method is reflected in the following example.

Example 4.1: *Let currentTheoremLhs be the left hand side of the currently analysed equation and fs-sort a build-in function returning the sort of a function-symbol.*

```
NAME                method1
DESCRIPTION         Generation of the left hand side of a new
                    theorem newTheorem from the recursive case
                    of ?sort (generalize each recursive
                    occurrences of ?sort in the right hand side
```

[2]The types of conflict information presented here are obviously not very refined. A refinement of the conflict information is part of our future work.

```
                    of the recursive case by a new, distinctive
                    variable
TYPE                static
PRECONDITION        Sensor_1 = true (?sort)
CONFLICT INFO.      planning(currentNode)
PROGRAM             t = fs-rec-rhs(?sort)
                    m = fs-occur(?sort,t)
                    for i=1 to m do
                       l_i = var-gen
                       t_i = fs-subterm(t,?sort)
                       t = fs-replace(t,?sort,l_i)
                    endfor
                    newTheoremLhs = ord(t)
                    newTheoremRhs = and_i^m(ord(l_i))
                    nd-attach(currentNode,
                              nd-new(newTheorem,n,Method_1,none))
POSTCONDITION      maybe-verifiable(currentNode)
```

4.2 Generation of Methods

In this section we describe how new methods in PLATIN are created. Basically this will be done in a Lisp environment in which a macro named (defun-method) allows the generation of new methods. In the moment of creation, i.e. evaluation of the defun-method expression, the method will become part of PLATIN. The components of a method, as described above, are: name, description, type, precondition, set of sensors, conflict information, postcondition and program. This enumeration coincides with the parameter list of the macro (defun-method). With the help of the example of the previous section, the use of the macro (defun-method), in order to create and add a new method to PLATIN, is illustrated.

Example 4.2: *Line 1 contains the method's name 'method1', lines 2,3,4,5 the description, line 6 its type 'static', line 7 the list of sensors '(sensor1) which contains just one sensor that will be referenced by the precondition in line 8, line 9 the conflict information, line 10 the postcondition and ultimately the lines from 11 to 24 the program of the method.*

```
1 (defun-method method1
2 "Generation of the left hand side of a new theorem newTheorem from the
3  the recursive case of ?sort generalize each recursive occurrences of
4  ?sort in the right hand side of the recursive case by a new,
5  distinctive variable"
6   ('static)
7   ((sensor1))
8   ((and (eq sensor1 'true) (bound? sensor1.?sort)))
9   ((planning (current-node (get-conflict-manager))))
10  ((maybe-verifiable (get-current-node)))
11  (let* ((?sort (value sensor1.?sort))
12        (term (fs-rec-rhs ?sort))
13        (m (fs-occur ?sort term))
14        (new-var nil)
15        (term-list nil)
```

```
16        (new-theorem nil)
17        )
18    (dotimes (i m)
19           (setq new-var (var-gen))
20           (push (list 'ord new-var) term-list)
21           (setq term (fs-replace term ?sort new-var))
22           )
23    (setq new-theorem (make-equation (list 'ord term) (and-n term-list)))
24    (nd-attach (nd-new new-theorem :n 'method1 'none))))
```

There are two possibilities to insert a new method into PLATIN. Firstly, the respective macro can directly be loaded into the LISP environment. Secondly, it is possible to create a file that only consists of method macros. This file can then be included into PLATIN from the user-interface, see appendix A. When PLATIN is loaded, a set of default methods is provided that can be found in $(PLATINHOME)/methods/methods.platin. The user is allowed to adapt this file to his own wishes, adding or deleting method macros.

5 Planning Tree

An essential part of PLATIN is the structure of a planning tree, which represents a plan. A node of a planning tree consists of a conjecture and control information. The control information is split into a range of characteristics, each of which is represented in the node by a respective slot and its value. In the next section we describe the node structure, its slots and their influence on the planning process. Afterwards we introduce a representation for planning trees to increase their readability. Then follows a section which gives a definition of a consistent planning tree and that of a complete planning tree. A complete planning tree can be seen as the ultimate goal of the planning process, as it guarantees that there exists a proof for the correctness of the root's conjecture. This guarantee can only be given if the planning trees generated during the planning process are consistent. To such an end the so-called propagation has been devised which transforms a planning tree, that has been derived from a consistent one by changing one of its nodes, to a consistent planning tree.

5.1 Node Structure

CONJECTURE (:c)	s = t
POSITION (:p)	1, 1.1, 1.2, 1.1.1, 1.1.2, ...
PROOF-STATE (:ps)	open, verified, rejected, locally verified, locally rejected
PROOF-TRIED (:pt)	yes, no
USED (:u)	yes, no
GENERATED-BY (:gb)	<method-name>
ADDITIONAL-INFORMATION (:ai)	maybe-verifiable, insufficient, verifiable, ...
FORBIDDEN-METHODS (:fm)	<method-name>*
NOTEBOOK (:nb)	(<method-name> <value>*)*

Figure 2: Node Structure

The node of a planning tree consists of a conjecture (an equation) and control information, see figure 2. The control information comprises eight slots. *Position* is an ordinal which presents a unique identifier of each node. The *proof-state* of a node reflects whether the node's conjecture is open (initial state), verified, rejected, locally verified or locally rejected. Open implies that the verifier has not proven or rejected the conjecture or that no proof has been tried, yet. If the proof-state of a node is verified (rejected) the verifier was able to prove (reject) the conjecture with the existing verified children. This is to say, that the conjecture is definitely correct or not correct. Note, that a node of which the proof-state is verified (rejected) is also called a verified (rejected) node. In the case of an open proof-state it is not obvious whether the verifier has tried to verify the node's conjecture or whether no verification step concerning the node has been activated, yet. To allow a distinction between an initially open node and an open node, for which verification has been tried, the *proof-tried* slot has been introduced. Note, that this information is of importance for the conflict manager when choosing between a planning or verification step, see also section 6.

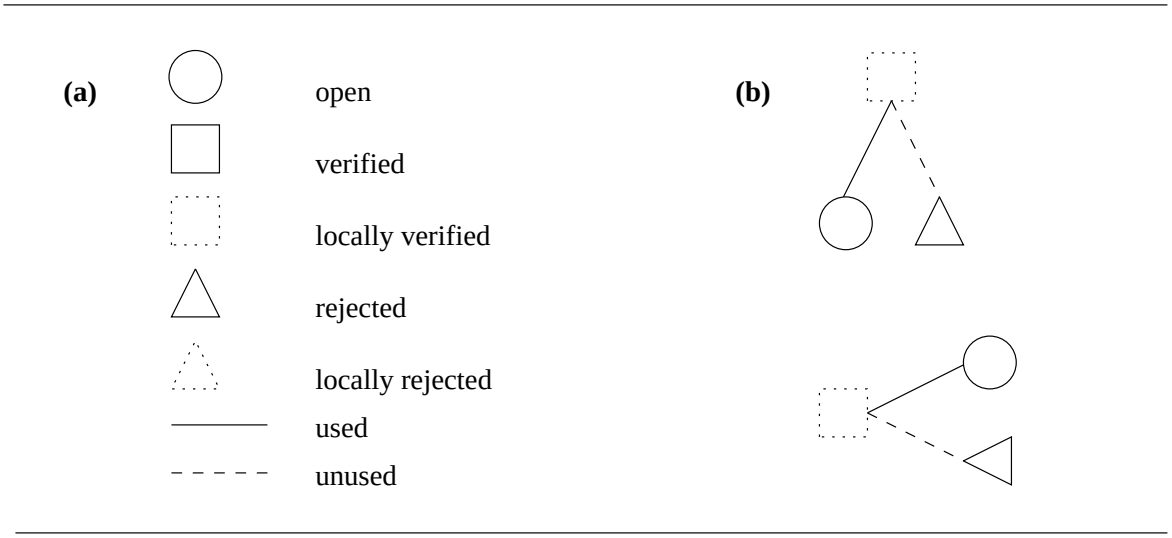

Figure 3: Representation of planning trees

The slot *used* describes a relation between the respective node and its parent node. The initial value of the slot used is 'no'. It will be set to 'yes' if a verification step concerning the parent's conjecture has been performed and the child node's conjecture has been applied in this verification process as a lemma. Nodes of which the slot used is set to 'yes' are also called used nodes. The *Generated-by* slot identifies the method by its name, which has led to the generation of the respective node. The postcondition of this method is stored in *additional-information*. *Forbidden-methods* gives the names of those methods which are not to be applied to the respective node. A method name will be included in the forbidden-methods of a node if it has been applied to this node and a further application of the method will not lead to new decompositions, see also section 4.1. In the node's *notebook* every applied method can put down or obtain notes during application. A note is associated with it's owner, i.e. the name of the corresponding method. A method can thereby only access its own notes. This feature is of special interest to both types of methods, i.e. static and dynamic ones, in order to store some 'internal' results as notes during their application. Due to the fact that dynamic methods can be applied several times in a planning process, a dynamic method can obtain notes from earlier applications and this allows such a method to control its own termination, as required. Note, that the values of forbidden-methods as well as of proof-tried and of notebook are not revealed to the user, as they are parameters intended for internal use, only. Let us now take a closer look at the information flow between the verifier and the planning tree. In case the conjecture of a leaf is to be verified, only the conjecture will be given to the verifier. For the verification of an inner node the verifier will receive, in addition to the current node's conjecture, all the conjectures of direct children. The conjectures of the children are treated as lemmata, called additional lemmata, by the verifier, which can then be used during the verification of the parent's conjecture. Note, that only children of which the conjectures are neither rejected nor locally rejected are regarded as additional lemmata. This implies that conjectures can be used as lemmata that, at a later stage, are proved to be incorrect. After a verification run the verifier provides information on whether the conjecture was verified or rejected and which of the children was used in the verification. This information is then stored in the proof-state slot of the current node and the used slot of the children, respectively. If the verifier is capable of proving (rejecting) the conjecture of the current node without using any of the additional

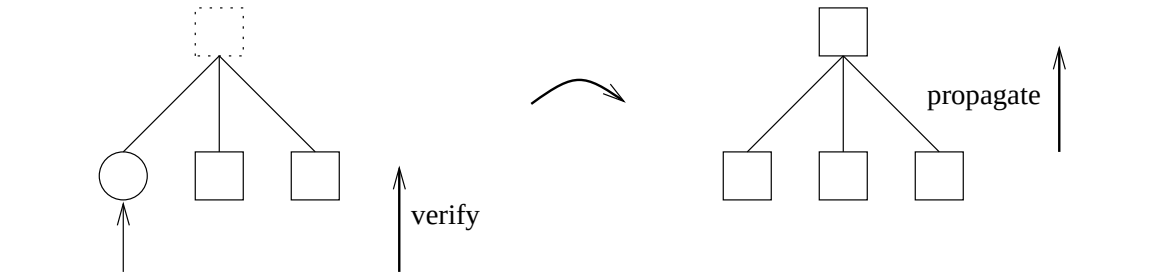

Figure 4: Propagation I

lemmata, the proof-state of current node is then set to verified (rejected). This is also the case if only verified children were employed during the verification. If, on the other hand, the verifier has used at least one non-verified child in the proof (rejection), the proof-state is set to locally verified (locally rejected). A verification run can also end without verifying or rejecting the current conjecture. In this case the proof-state remains open and proof-tried is set to 'yes'. Currently, the verifier interrupts a proof attempt whenever a certain number of proof steps (adjustable by the user) have been performed without a definite result, i. e. without proving or rejecting the conjecture of the current node.

5.2 Representation

To increase the readability of planning trees in the following illustrations, we introduce a suitable representation. The most important slots of a node in the following figures are the proof-state slot and the used slot. Therefore the corresponding slot values will be represented by graphic elements like circles, squares and so on. As can be seen in figure 3, a node with proof state 'open' will be depicted by a circle, a node with proof state 'verified' by a square and a node with proof state 'rejected' by a triangle. A dotted square (triangle) will be used for the state 'locally verified' ('locally rejected'). The value of the used-slot will be given by means of solid and dashed lines, where a solid line stands for 'used' and a dashed one for 'unused'.

In the following a planning tree will be presented in two ways (cf. figure 3 (b)). Commonly, a tree is shown with the root at the top and the leaves at bottom positions. In some cases it is reasonable to show the tree with the root at leftmost and the leaves at rightmost positions. Take for example a tree, in which the nodes form a chain. Every tree induces a set of positions for the identification of nodes, as usual. In the subsequent figures we will only refer to these positions when necessary.

5.3 Propagation

As mentioned at the beginning of this section a change of one proof-state has consequences for other proof-states or, in other words, the change has to be propagated. To show the necessity of this propagation, we present, first of all, a few typical examples. Then we define a consistent planning tree and show how propagation, which transforms an inconsistent planning tree into a consistent one, works. Note, that the following examples feature the representation introduced in the preceding section 5.2. Figure 4 shows a planning tree consisting of a root, which is locally verified, and three children. One of them is open, the others are verified. The former is verified by a verification step. The change of the child's

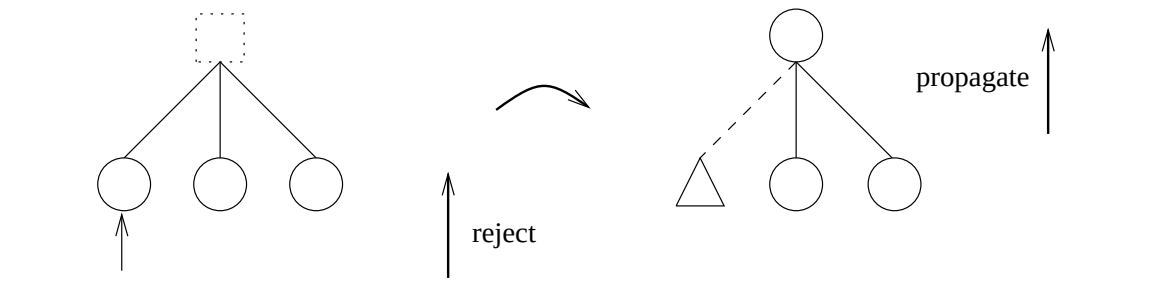

Figure 5: Propagation II

proof-state from open to verified has to be propagated to the root. This is done by changing the proof-state from locally verified to verified. Figure 5 shows another planning tree. Similar to the first example the root is locally verified. All children are open and used. One child will be rejected after a verification step. This means the root was proven to be correct with an additional lemma, which is now shown to be incorrect. This, in turn, is propagated to the root by setting the root's proof state back to open. Moreover, the proof-tried slot is set back to 'no', the slot for additional information is reset to the initial value and the rejected child will be marked as unused. Resetting the proof state allows the execution of a further verification step for the root. This is necessary as the former verification process has lost its validity as one of the applied lemmata proved to be incorrect. Initializing the root's additional information allows further planning steps on the root.

To further clarify the way propagation works, we give a definition of a *consistent planning tree*. Leafs can feature the following proof-states: open, verified and rejected. The proof-state of all nodes is initialized to 'open'. The proof-state is open if and only if no or an unsuccessful proof has been tried. It is verified or rejected if and only if all children, which were used in the appropriate verification, are verified. It is locally verified or locally rejected if and only if at least one used child is not verified. A planning tree fulfilling these conditions is called to be consistent.

We can now present a description of *propagation*. The change of one proof-state can possibly affect all the nodes on the path up from the respective node to the root. Hence the following procedure has to repeated until reaching the root. Consider now a node n and its possible parent n'.

1. If no parent n' exists then n is the root and the propagation stops.
   If n is not used (:u no) or the proof-state of n is set to 'open' or 'locally verified' then the propagation stops.

2. If the proof-state of n is set to 'verified' and the proof-state of n' is 'locally verified' and the proof-state of all other used children of n' is 'verified' then change the proof-state of n' to 'verified'. Set n to n' , n' to the father of n' and goto 1.

3. If the proof-state of n is set to 'verified' and the proof-state of n' is 'locally rejected' and the proof-state all other used children of n' is 'verified' then change the proof-state of n' to 'rejected'. Set n to n', n' to the father of n' and goto 1.

4. If the proof-state of n is set to 'rejected' or 'locally rejected' then change the proof-state of n' to 'open', initialize proof-tried and additional-information of n' and set used of n to 'no'. Set n to n', n' to the father of n' and goto 1.

6 Conflict Manager

The part of PLATIN which coordinates the planning and verification process is called conflict manager, as illustrated in figure 6. The conflict manager activates methods, sensors and the verifier. In addition, the conflict manager informs the user about the planning state and supports the user in the non-automatic mode. The first section describes the principal procedures according to which the conflict manager works. They will be performed in a loop. This loop reflects the sequential actions like choosing a node, deciding on refinement of the plan, choosing and activating methods or activating the verifier. The second section concentrates on general planning strategies that the conflict manager can follow. These planning strategies describe how verification and planning steps can be combined. In addition, the planning strategy currently realized in PLATIN will be stated. The third section will give a more natural description of an important heuristic, the heuristic to decide on a node with which to proceed in the planning process. The efficiency of the whole planning process and its success strongly depend on this heuristic. The last section describes the heuristic concerning the choice of refinement or verification of the plan. This essential heuristic affects the number of planning steps and hence the efficiency of PLATIN.

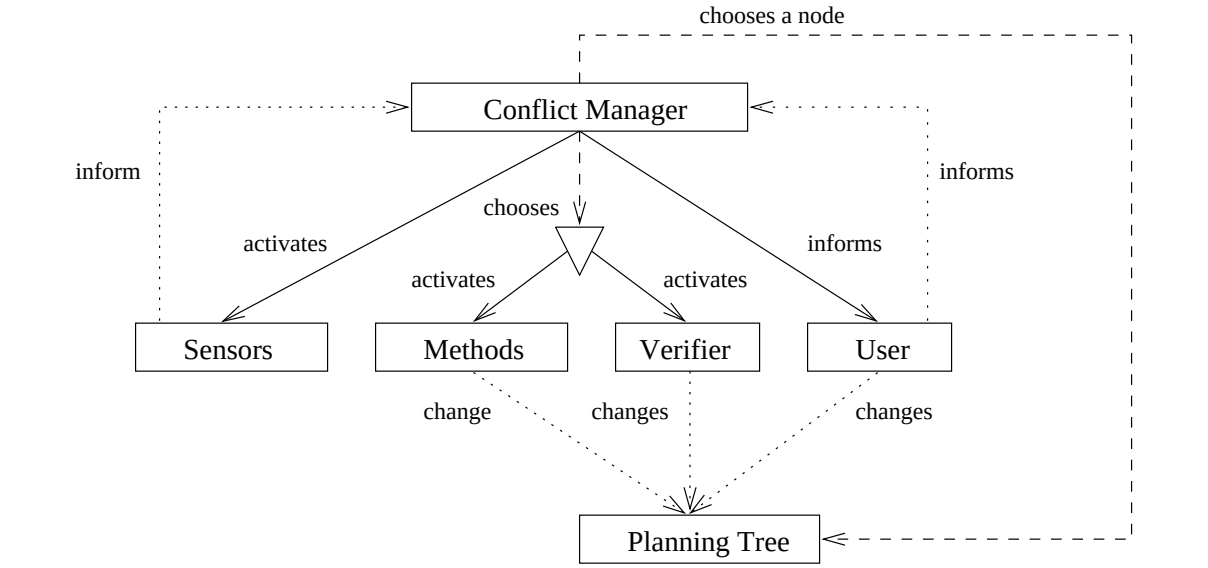

Figure 6: Conflict Manager

6.1 Main Procedure

This section describes the main procedure of the conflict manager. An activation of the conflict manager leads to several actions which will be performed according to the loop illustrated in figure 7. First the conflict manager starts a heuristic search for a node to work on. The heuristic underlying this decision will be described at a later stage. Afterwards, all sensors will be activated in order to store measurements of the system in its variables. The preconditions of the methods are related to these measurements and allow a decision whether a method is applicable or not. Then the conflict set, i.e. the set of all applicable methods, will be computed by evaluating all preconditions of the methods. In the case of a

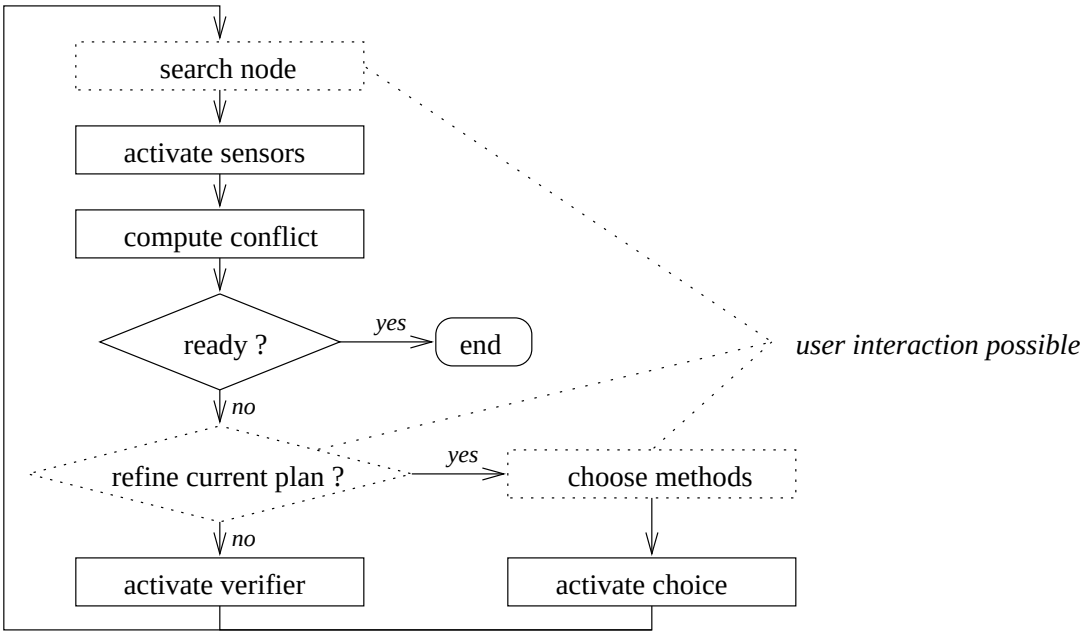

Figure 7: Main Procedure of the Conflict Manager

complete or rejected plan, i.e. the root is verified or rejected, the conflict manager will stop and inform the user about this result. Otherwise the conflict manager heuristically decides either on refinement or verification of the current plan. This heuristic named 'Refinement-or-Verification' will be described later on in a more precise manner. In the case of refinement one method has to be chosen from the conflict set. In this case every conflict information has to be evaluated in order to measure the conflicting methods by weights. The conflict manager then chooses the most promising one according to these weights, if possible. If such a decision can not be made, the conflict manager requests the user to choose one or more methods out of the conflict set. Afterwards he activates the chosen methods. If the conflict manager wants to perform a verification step, the verifier will be activated with the current node. The verifier can use the direct children of the current node as additional lemmata, more precisely the not rejected and not locally rejected children. The result of the corresponding proof determines the new proof-state of the current node, see also section 5.

6.2 Planning Strategies

The basic idea of PLATIN is to split a problem into several subproblems and to repeat this process until the subproblems can be proved by the verifier. However, the order in which the two actions are performed can be manifold. Planning strategies describe the varying combinations of these actions.

If a node can be proven with his direct children as additional lemmata, the respective decomposition is called safe. This implies several possibilities for the way of planning. One is to prove the children before proving the parent, another to prove the parent with his children and then proving the children. We will briefly discuss the advantages and disadvantages. The former, as illustrated in figure 8 a), leads to repeated unsafe decompositions of the problem until no further decompositions are possible. Then the verifier starts to prove the nodes starting at the leaves of the planning tree and thereafter moves up to the root. In

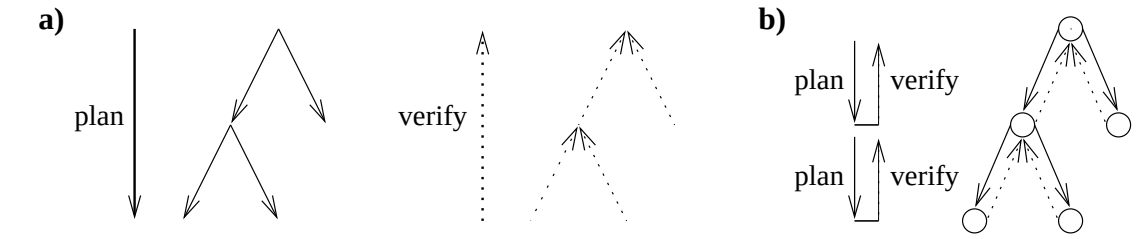

Figure 8: Strategies

this case, i.e. to plan as far as possible and only then to start verification, the planning and the verifying process are not interleaved. The advantage is that only open, verified or rejected nodes occur. This way locally verified or locally rejected nodes, which represent 'unsafe knowledge' and are more difficult to handle, can be avoided. The disadvantage is risking a decomposition which can not be shown as safe by the verifier. Especially if the root can not be proven with its children, a lot of work and time has been wasted. Furthermore, if a decomposition is shown to be safe, possibly just a few children were used in the corresponding proof. In this case, only the corresponding subtrees in the planning tree are of importance, the others can be removed. This, in turn, implies that the work performed on subtrees of unused children was unnecessary.

The second strategy, i.e. to prove the parent with his (direct) children and then proving the children, leads to an alternation between planning and proof steps. Figure 8 b) gives an illustration of this strategy. An unsafe decomposition is immediately followed by a proof step to guarantee the safety of the decomposition. This process stops when no further decomposition is possible or a complete tree has been achieved. The advantage of this method is to immediately ensure whether the parent is provable with his children or not. In addition the children which are useful in further planning steps, i.e. the children which the verifier has used in the respective verification process of the parent node, can be identified at once. This information is returned by the verifier at the end of a verification step and will be inserted in the corresponding slot of the children's nodes. The disadvantage is frequent work with unsafe knowledge, i.e. many nodes are locally verified or locally rejected. Take for example the following situation: the decomposition is a generalization, i.e a node becomes a generalized child. Normally, the verifier can prove the parent with the generalized child, even though the generalized child can also be wrong. In this case the parent, which can be correct or not, is still locally verified. In general, it can happen, that a method creates unsuitable children for a node and the verifier can prove the node with its children, i.e. the node is locally verified. This process can be repeated until the verifier tries to prove a leaf and possibly rejects the leaf. At this stage expensive replanning would be necessary.

Both strategies are extremes in the way of planning. In our opinion the most promising strategy lies between the two extremes, which represent opposite standpoints in the way of planning. This can clearly be seen in the advantages and disadvantages of both strategies. The first has the advantage of 'safe knowledge' and the disadvantage of 'unsafe decompositions'. In contrast, 'unsafe knowledge' is the disadvantage of the latter and 'safe decompositions' its advantage. A compromise is realized in PLATIN with the help of the methods' additional information, which is stored in the appropriate nodes. This additional information supports the conflict manager in his decision on refinement or verification.

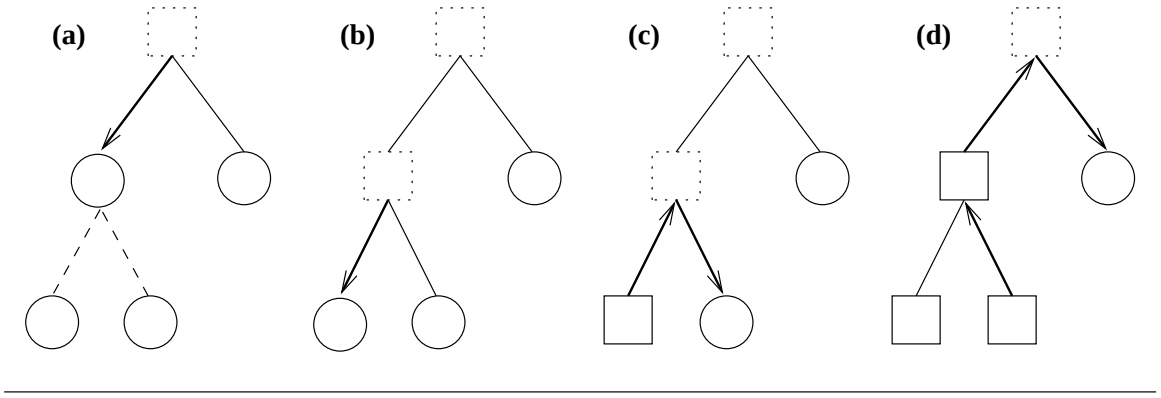

Figure 9: Choose Node

When a node is marked as 'insufficient', which implies the method considers the verifier unable to prove this node with its children, then a further refinement step is advisable. On the other hand, if a node is marked as 'maybe-verifiable', then the method has stated, that this node is possibly verifiable with his children. This additional information affects the decision on refinement or verification of the plan and yet allows to realize both extreme strategies. Marking all nodes as 'insufficient' leads to the first strategy, marking all nodes as 'maybe-verifiable' to the second one. In practice a strategy between the two extremes will be achieved, that is the compromise mentioned above. This strategy actually features dynamic behaviour because of its dependence on additional information and hence on the changeable set of methods.

6.3 Heuristic Choose-Node

PLATIN's search heuristic for choosing a node is very similar to the well-known Depth-First-Search, starting with the root as can be seen in figure 9, then going down until a leaf is reached and thereafter going up until another way down is possible. Because of this similarity we do not give an exact definition of this heuristic, but some distinguishing situations, only. Consider the situation given in figure 10. The subtree T has a root, which

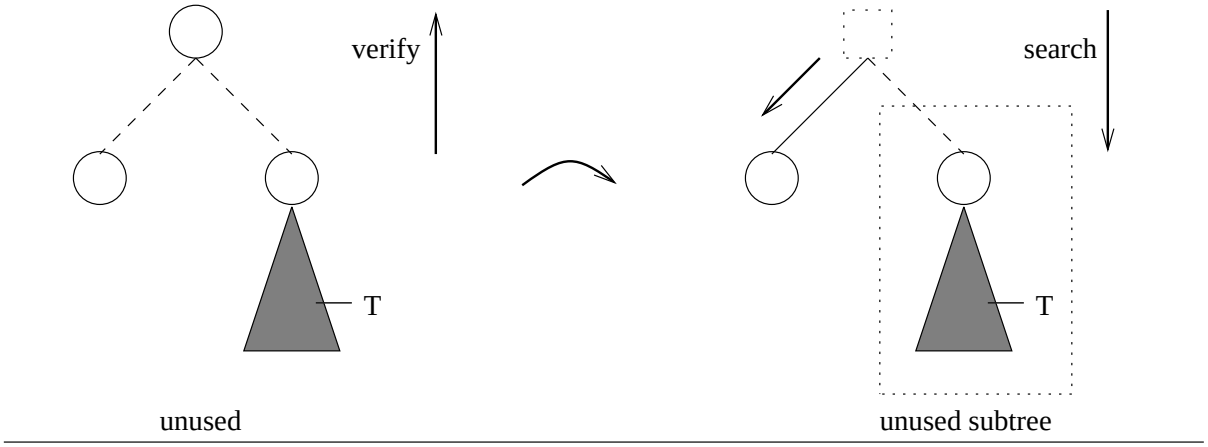

Figure 10: Avoiding unused Subtrees

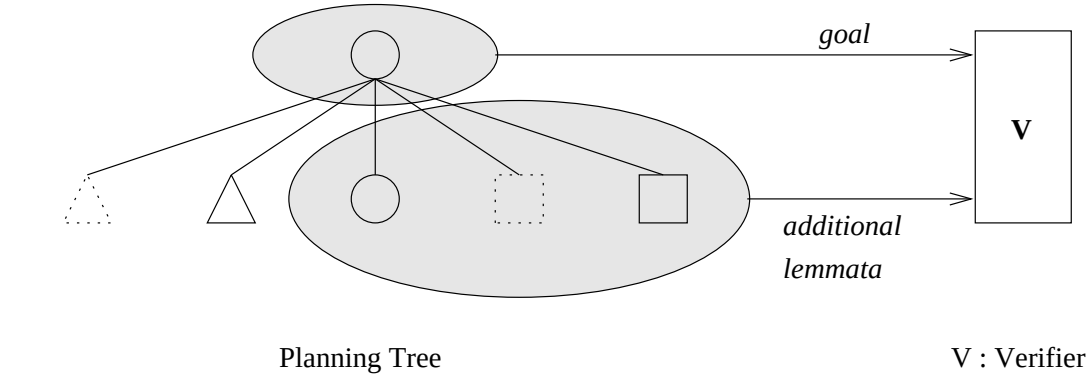

Figure 11: Avoiding locally rejected or rejected Subtrees

is open and unused. After the verification step, the root of the whole tree can be proven and is hence locally verified. But only the left child was used in the corresponding verification. During future activations of the search-heuristic, the nodes of subtree T will no longer be considered except if the root's proof-state is changed to 'open'. Therefore, further work on nodes within T will not be performed. However, verified nodes within T could be useful in further verification steps, perhaps as additional children of the root. For this purpose a dynamic method could look deeper into T [3] to collect all verified nodes and to expand the root with these nodes. Note, that the set of additional lemmata and especially its size has a strong influence on the success of the verification. Experience shows, that neither too few nor too many additional lemmata are positive in the verification. The former naturally follows while the latter is not immediately to be derived. Most verifiers use heuristics to manage their verifications. For example, heuristics for choosing an inference rule (induction, case distinction, simplification, ...), heuristics for choosing a induction variable, etc. If a verifier can prove a conjecture with a set of lemmata L, it does not imply that the verifier is successful with a greater set $L' \supset L$ of lemmata. Take for example a simplification, which transforms the conjecture in a 'simpler' conjecture with the help of a lemma from L. Another lemma, a lemma from L' for example, can lead to another 'simpler' conjecture, which the verifier now isn't able to prove. Therefore, it is often not helpful to collect all verified nodes within T with a static method and then to expand the corresponding node with these nodes. Summarizing, the search-node heuristic of PLATIN regards only used children and the corresponding subtrees. This is why the whole subtree T in the example will not be considered by our heuristic.

Figure 11 illustrates which direct children are used by the verifier as additional lemmata. In the situation described in the figure, the root is the conjecture and all not locally rejected and not rejected children are additional lemmata. Note, that a locally rejected child can still be correct. Locally rejected means, that the verifier can reject this child with its appropriate direct children and at least one used child is not verified. This will be described more precisely in section 5.1. It is evident not to use a rejected child as additional lemma, but not to use locally rejected children needs further explanations. The decision not to use locally rejected children is a heuristic one, based on the assumption that a locally rejected subtree, i.e. a subtree with a locally rejected root, is likely to be rejected later on. If such

[3] Only a dynamic method is able to be applied several times on one node.

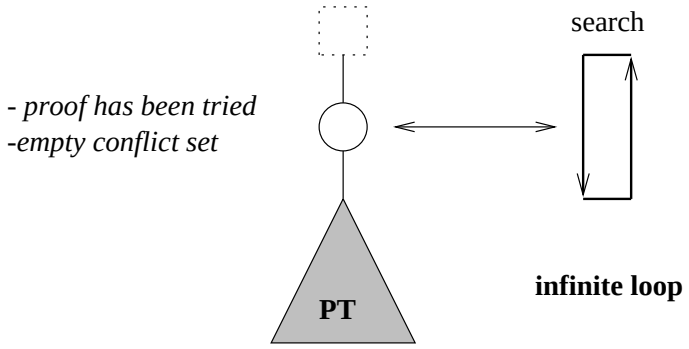

Figure 12: Infinite Search Loop

a locally rejected child would have been used as an additional lemma this would then lead to expensive replanning. On the other hand a locally verified subtree is assumed to be a tree which is likely to become a verified subtree with further appropriate proof and planning activities. Naturally, this heuristic can fail like the others described in the preceding sections.

Figure 12 shows a critical situation, which has to be avoided by the search heuristic as it would lead to an infinite search loop. Starting with the locally verified parent, the search heuristic goes down to its only child, which is open and used. Furthermore, a proof has been tried and the conflict set is empty, i.e. no other method is applicable. Hence, a refinement step for this child is not possible. On the other hand a verification would lead to the same result, namely the verifier can neither prove nor reject the child. This is why it would seem reasonable for the search heuristic to go up, back to the parent. This, however, would lead to an infinite search loop. To avoid this critical situation, the search heuristic realized in PLATIN gives a warning and some user hints on how to change this situation. One possibility is to increase the verifier steps, this would lead to a new (deeper) proof. Another possibility is either to add new children [4] or to load new methods, which are possibly applicable in this situation. The former allows a new proof, the latter might lead to a refinement step and, hence, afterwards to a new proof.

6.4 Heuristic Refinement-or-Verification

As can be seen in figure 7 the conflict manager has to decide whether to verify or to refine the current plan. This decision is based on a heuristic, which will be described in this section. We will give the description in a more informal manner. The following figures contain a decision tree, representing the underlying framework of this heuristic. Every edge in this tree is labeled with a (possibly empty) condition. The outgoing edges of a node, more precisely the disjunction of the appropriate conditions, represent a complete, exclusive case distinction. The leaves are either associated with decisions, e.g. 'refine' or 'verify', or with errors or warnings, indicating an inconsistence of the planning tree or an useless node. When detecting a verified node [5], on which no proof has been tried, the heuristic signals an error. In this case the planning tree is inconsistent. Note, that strong methods also set the proof-tried slot to 'yes' when guaranteeing the proof state ':v' for a node.

[4] A refinement step performed by the user.

[5] i.e. a node with proof-state ':v'

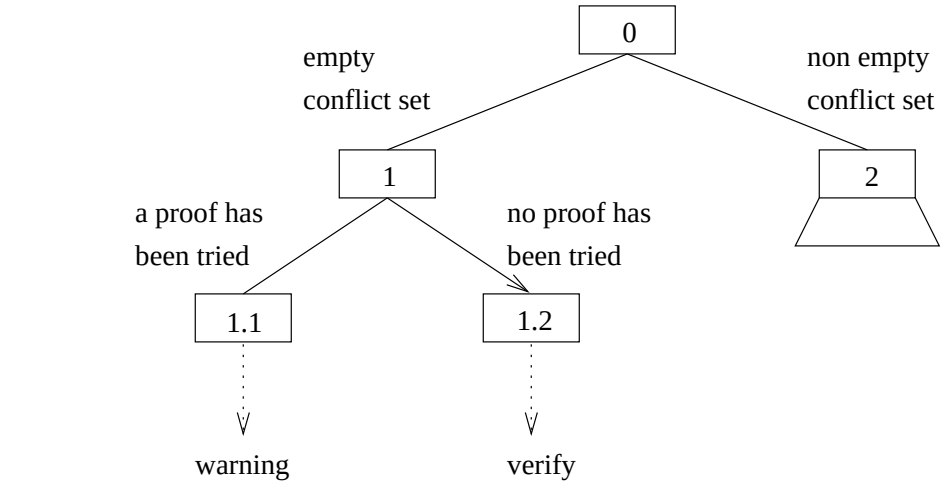

Figure 13: Decision Tree I

When analysing a verified or rejected node in regard to planning, the heuristic issues a warning. In such a situation a decision for refinement or verification would be senseless. In a given situation the heuristic follows the edges according to the evaluation of the corresponding conditions and returns the decision associated with the appropriate leaf. Note, that every situation induces a definite path in the decision tree due to the complete, exclusive case-distinctions mentioned above. In the following, the nodes of the decision tree will be called states in order to differentiate between these nodes and nodes of the planning tree.

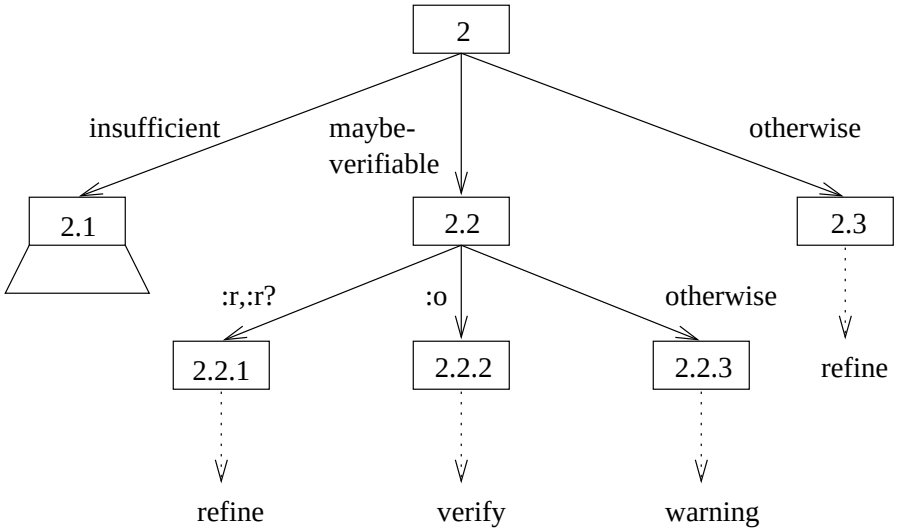

Figure 14: Decision Tree II

At the beginning (state 0) the heuristic determines the conflict set (cf. figure 13). If this set is empty (state 1) the heuristic checks whether a proof has been tried on the current node or not. When at least one proof has been tried (state 1.1), the heuristic gives a warning, otherwise (state 1.2) a verification will be performed. State 1.1 is associated with a warning

to avoid termination problems. Due to the empty conflict set a refinement step with the existing set of methods is impossible. Because a proof has already been tried, a verification step is only reasonable with new verifier-parameter, e.g. with a higher number of possible verifier steps. At state 2, i.e. there exists at least one applicable method (cf. figure 14), the heuristic looks at the additional information given in the current node. If the node is marked as 'maybe-verifiable', then the heuristic determines the proof state. In case of a rejected or a locally rejected node (state 2.2.1), the heuristic decides to perform a refinement step, that is, the existing decomposition at this node will be replaced by a new one. In case of an open node (state 2.2.2), the heuristic decides to verify the node, following thereby the hint given in the additional information slot. Otherwise (state 2.2.3), the heuristic gives a warning, because the current node is verified or locally verified, which implies that neither verification nor refinement on this node is reasonable. If the node is neither marked as 'maybe-verifiable' nor 'insufficient', then the heuristic decides to perform a refinement step.

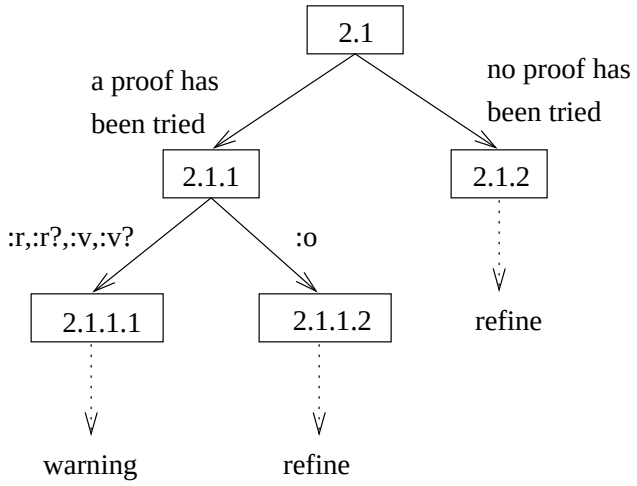

Figure 15: Decision Tree III

At state 2.1 the heuristic checks, whether a proof has been tried or not (cf. figure 15). If a proof has been tried, the heuristic examines the proof state of the current node. The proof states rejected (:r), locally rejected (:r?), verified (:v) or locally verified (:v?) lead to state 2.1.1.1, which is associated with a warning. The proof state open (:o) leads to state 2.1.1.2 , which is associated with a refinement-step. Note, that in this situation a further verification step is only reasonable with new verifier parameters (cf. section 7).

7 Verifier

Verification and planning are central parts of a proof process, which can be interleaved in PLATIN as described in the preceding sections. Both parts need a verifier, the former to ensure the safety of a decomposition, the latter to prove leaves of a planning tree.

In PLATIN the verifier will be seen as a black box, which requires some specific input and returns a particular result. In the following we call this black box *abstract verifier* and explain its input and result. Every concrete verifier has to respect the requirements depicted by the abstract verifier concerning input and result.

To clarify the way the abstract verifier is integrated in the proof-process, we generally describe its *method* of working in the PLATIN environment. To such an end we present a model illustrating the interaction between the conflict manager, the planning tree and, of course, the verifier.

The current implementation supports a single verifier, namely *UNICOM*. A detailed description of this verifier can be found in [GrLi91].

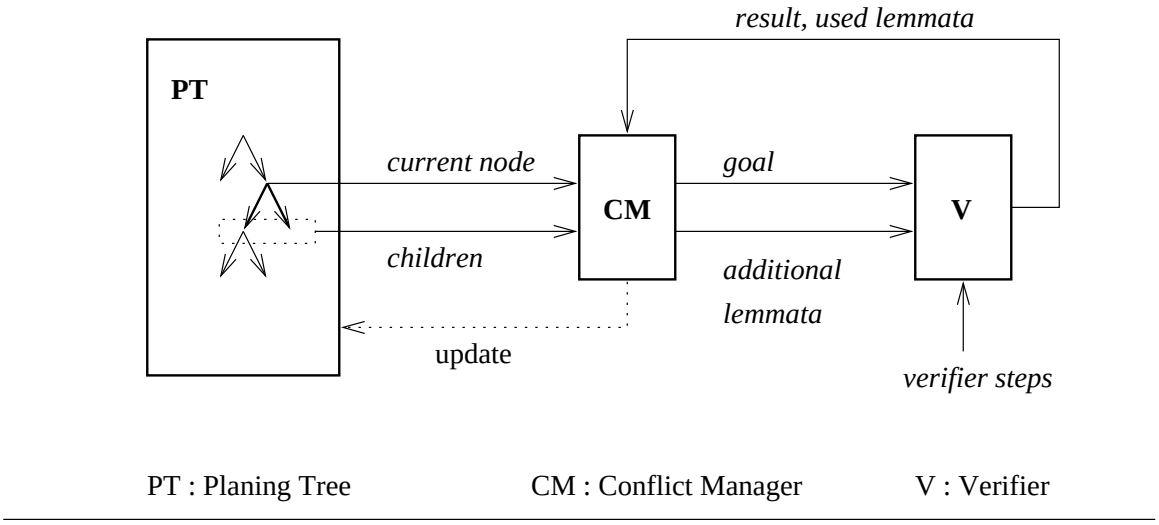

Figure 16: Verifier

7.1 Abstract Verifier

Before a description of the abstract verifier we give an overview of the logical background of PLATIN. PLATIN supports many-sorted first order predicate logic with equality, so it can be used for equational logic just as well as for Horn logic. In our framework a given specification Φ, i.e. a set of sentences over a given signature, is associated with a semantic $sem(\Phi)$. The semantic is a class of structures, which are all models [6] of Φ. To ensure the correctness of PLATIN, we require some kind of deduction theorem.

$$sem(\Phi \cup \{\varphi\}) \models \psi \text{ iff } sem(\Phi) \models \varphi \rightarrow \psi \tag{1}$$

where φ, ψ are sentences over the given signature. This can be seen as the semantical foundation of the propagation described in section 5.3. In the following we briefly sketch

[6] in the ordinary sense, cf. [EFT92]

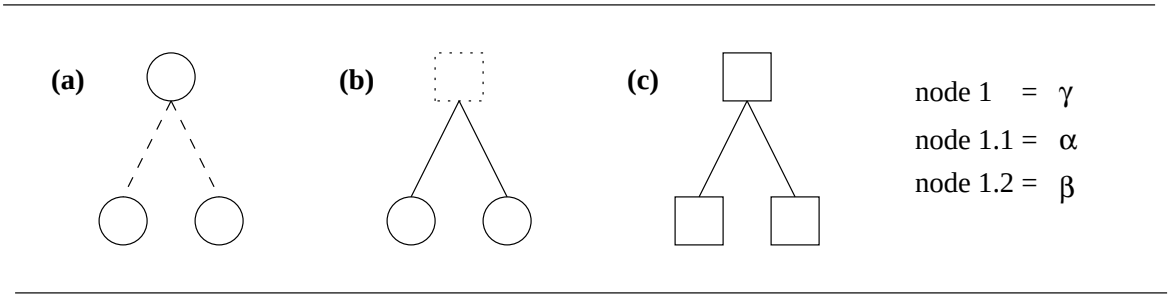

Figure 17: Example

this by means of an example. In the following a verifier V will be seen as a 2-placed relation $\vdash_V$ for which we require the following correctness property:[7]

$$\Phi \vdash_V \psi \text{ implies } sem(\Phi) \models \psi \tag{2}$$

For strong methods we require a similar correctness property. Let M be a strong method, which is applicable (the precondition of M is fulfilled) on node n and will create the new children $n_1, \ldots, n_k$. Let ψ the conjecture of node n and $\psi_1, \ldots, \psi_k$ the conjectures of nodes $n_1, \ldots, n_k$ respectively. Now we require:

$$sem(\Phi) \models \bigwedge\{\psi_i \mid i = 1, \ldots, k\} \rightarrow \psi \tag{3}$$

Example 7.1: *Let Φ a given specification. The planning tree in figure 17 (a) contains only open (:o), unused nodes. After a verification step (cf. figure 17 (b)) the root γ is locally verified and the children α, β are both used. This means $\Phi \cup \{\alpha, \beta\} \vdash_V \gamma$ and hence $sem(\Phi \cup \{\alpha, \beta\}) \models \gamma$. From (2) follows immediately $sem(\Phi) \models \alpha \wedge \beta \rightarrow \gamma$. After two verification steps (cf. figure 17 (c)) both leaves are verified (:v). Propagation leads to the root's proof state 'verified'. The correctness follows from $\Phi \vdash_V \alpha$, $\Phi \vdash_V \beta$, which implies $sem(\Phi) \models \alpha \wedge \beta$ and hence $sem(\Phi) \models \gamma$.*

Now, we can describe the abstract verifier (AV for short). The AV takes a goal and a set of additional lemmata as input. The AV has access to the specification and the already proved conjectures at any time. Furthermore, the proof depth of the AV can be adjusted by a natural number, called verifier-steps. This number is also an input parameter. By requiring such a restriction of the proof depth the termination of every verification process is guaranteed. In case of an unsuccessful proof, i.e. the verifier could neither verify nor reject the conjecture with the given verifier-steps, an augmentation of the verifier steps has to lead to another, deeper proof. The output of the AV does not only consist of the proof-result, i.e. verified, rejected, open (verification interrupted after the specified proof depth without a specific result), but also of the set of the additional lemmata given as input, which were used in the corresponding verification process.

7.2 Verifier Method

Figure 16 shows a principal verification step involving the components planning tree, conflict manager and verifier. After deciding to perform a verification step, the conflict manager determines the goal, which is the theorem of the current node, and the additional lemmata,

[7] A tuple $(\Phi, \varphi) \in \vdash_V$ consists of a set of sentences Φ and a single sentence φ. Conventially $(\Phi, \varphi) \in \vdash_V$ will be written as $\Phi \vdash_V \varphi$.

which are the used, not rejected or locally rejected children of the current node. Goal and lemmata will be given to the verifier, which performs a verification process restricted by an upper bound of possible proof steps, the so-called verifier-steps or step-limit. This ensures the termination of the verification process. Additionally, the verifier specifies which of the additional lemmata were really used in the performed proof. The number of verifier-steps is one of the parameters, which can be directly adjusted by the user (cf. appendix A). It is obvious that the additional lemmata play an important role in the corresponding proof, therefore the verifier needs a possibility to employ these lemmata with a high priority. Currently PLATIN has UNICOM as an underlying verifier. UNICOM provides in its structure possibilities to give certain lemmata high priority. A slight modification and extension of UNICOM was needed to make these possibilities explicitly available. It should be mentioned, that verification and planning can not be seen totally independent from each other. A method which performs a successful decomposition has to generate useful hypotheses, i.e. hypotheses, which are useful for the verifier in the corresponding decomposition-proof. Therefore, such hypotheses often reflect some characteristics of the underlying verifier. This is why good methods, i.e. methods, which frequently participate in successful proofs of PLATIN, often contain implicit knowledge of the verifier.

7.3 UNICOM

UNICOM (*UN*failing *I*nductive *COM*pletion) is an inductive theorem prover for equational logic. It supports hierarchically structured many-sorted algebraic specifications. A specification consists of definitions of total functions and inductive conjectures, which UNICOM tries to prove or disprove. The main components of UNICOM are the following:

1. *parser*
   checks a hierarchic specification and generates an internal representation

2. *checker*
   checks the completeness and consistency of the function definitions

3. *prover*
   tries to prove or disprove a set of inductive conjectures

UNICOM supports a rule priority strategy for simplification: inductive rules will be applied before lemmata, lemmata before definition rules. Additionally, lemmata of 'higher' specifications have a higher priority as lemmata of imported specifications. This feature will be used in PLATIN when giving the additional lemmata, that occur in every verification step of PLATIN, the highest priority. Furthermore, every verification step of PLATIN will be started in a separate lisp process[8]. This allows the user interface and the verifier to work independently from each other.

[8]Some lisp implementations (not all) feature the ability to handle lisp processes.

8 Example

In this section we present a somewhat larger example. For this, we have chosen the well-known Quicksort algorithm. The specification of this algorithm, 'qsort', as well as a short description of all the mentioned methods and sensors can be found in the appendix (cf. appendix B). First, we describe the planning process, i.e. the different stages of development of the planning tree. Then we give a list of all the conjectures occurring in this example. Afterwards some typical user interaction is described, which can be required by the verifier during verification.

8.1 Planning Process

PLATIN is started with the following sensors and methods: sensor1, sensor2, sensor3, sensor4 and method1, method2, method3, method4. To plan the proof of the Quicksort algorithm its specification, called 'qsort', has to be loaded into the system. This will be done from the desk of the user interface which is described in the appendix (cf. appendix A). By default PLATIN works in automatic mode, which means that all heuristics described in chapter 'Conflict Manager' will be applied and that the system needs as little user interaction as possible. The chosen example will be illustrated by figures, that represent the planning tree in different states of the planning process. For simplicity, we sometimes refer to slot values of a node without explicitly mentioning the slot itself. For example, we say 'node n is verified', instead of 'the proof state of node n is verified'. We describe the planning process according to the main loop of the conflict manager (cf. section 6.1). First we explain the result of the search-node heuristic, then the decision of the refinement-or-verification heuristic. In case of a verification, we depict the corresponding proof result and its consequences to the planning tree. In case of a refinement, we present the conflict set and the method which will be activated.

At first, the planning tree has only one node, the root (cf. figure 18).

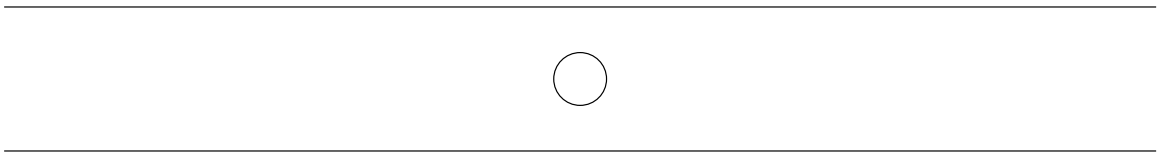

Figure 18: Example 1

Search Node: 1
Refinement or Verification : Refinement
Node 1 is open and no proof has been tried, hence a refinement step is reasonable.
Conflict Set : method1, method2
Activate : method1
Two methods are applicable, therefore the conflict manager evaluates the conflict information of method1 and method2. Method1 has more weight in this situation, so the conflict manager activates method1. After its application, the static method method1 will be inserted into the forbidden method list of node 1. From now on the conflict manager will automatically remove method1 from the conflict set at node 1. Application of method1 leads to the new node 1.1 and the additional information 'maybe-verifiable' for node 1.

Search Node: 1
Refinement or Verification : Verification
The proof-state of Node 1 is open and Node 1 has the additional information 'maybe verifiable'. Node 1 can be verified with node 1.1 as a used additional lemma, so the proof-state of node 1 is set to locally verified. Remember, that the verifier returns whether a additional lemma was used or was not used in the verification. Node 1 would get the proof-state verified, if node 1.1 were not used, for example. The current planning tree can be seen in figure 19.

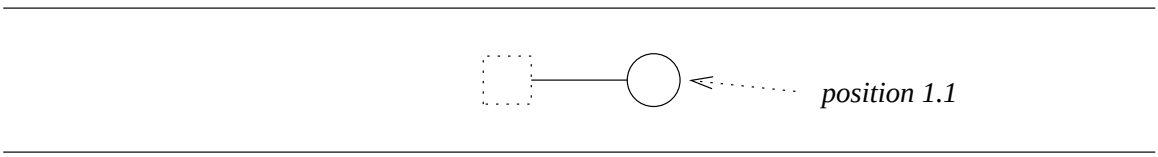

Figure 19: Example 2

Search Node: 1.1
Node 1 is locally verified, so the conflict manager chooses node 1.1.
Refinement or Verification : Refinement
Node 1.1 is open, no proof has been tried and the conflict set is not empty. Hence, a refinement step is chosen.
Conflict Set : method3
Activate : method3
The static method method3 creates node 1.1.1 and is inserted in the forbidden-method list of node 1.1.

Search Node: 1.1
Node 1.1 is open and no proof has been tried.
Refinement or Verification : Verification
No method is applicable, hence there is no other possibility as a verification step. Node 1.1 will be rejected with node 1.1.1 as a used additional lemma, so node 1.1 is locally rejected. Propagation of this state as described in section 5 leads to a change of the proof-state of node 1 from locally verified back to open. Further the slots 'additional information' and 'proof tried' will be set back to their initial values. Remember, that 'locally rejected' does not mean 'rejected'. In fact, node 1.1 could be true and 1.1.1 wrong. The current planning tree can be seen in figure 20.

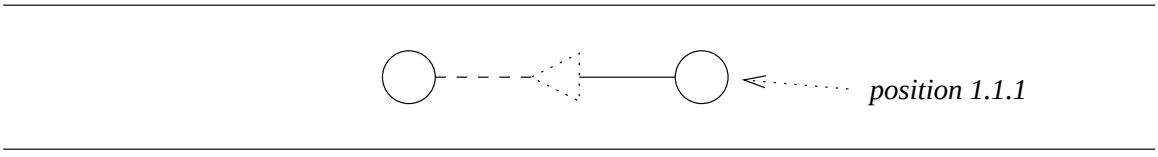

Figure 20: Example 3

Search Node: 1
Node 1.1 is locally rejected, so the conflict manager chooses node 1.
Refinement or Verification : Refinement
As node 1 is open, no proof has been tried and as the conflict set is not empty, the conflict

manager decides to refine the plan.
Conflict Set : method2
Remember, that method1 is also applicable but forbidden on node 1, i.e. an element of the forbidden-method list.
Activate : method2
Activation of method2 leads to the creation of following nodes : 1.2, 1.3, 1.4, 1.5, 1.6. Moreover, method2 marks node 1 as 'insufficient', which means that nodes 1.2,...,1.6 are possible not strong enough to prove node 1 and further refinement steps are necessary. The conflict manager takes this information into his consideration about refinement or verification.

Search Node: 1
Refinement or Verification : Verification
All applicable methods, i.e. method1 and method2, are forbidden on this node. Therefore, a refinement step is not possible, even though node 1 is insufficient. After the verification, node 1 is (again) locally verified but now with the help of nodes 1.2, 1.3, 1.4, 1.5 and 1.6 as used additional lemmata. Remember, that the locally rejected child 1.1 will not be regarded here as additional lemma. The current planning tree can be seen in figure 21.

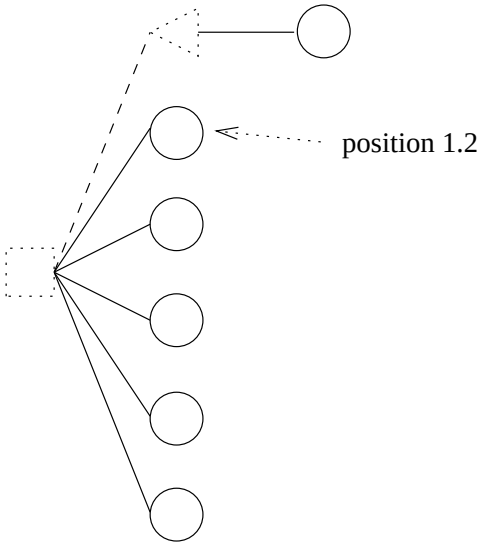

Figure 21: Example 4

Search Node: 1.2
Node 1 is now locally verified, so the conflict manager chooses node 1.2.
Refinement or Verification : Refinement
Node 1.2 is open, no proof has been tried and the conflict set is not empty. Hence, a refinement step is chosen.
Conflict Set : method3
Activate : method3
After its application the static method method3 is forbidden on node 1.2. Method3 creates node 1.2.1 and marks node 1.2 as 'insufficient', i.e. a further refinement of the plan is advisable when possible.

Search Node: 1.2
Node 1.2 is open and no proof has been tried.
Refinement or Verification : Verification
The only applicable method method3 is forbidden on this node, so the conflict set is empty. Node 1.2 can be verified with 1.2.1 as a used additional lemma and is thus locally verified.

Search Node: 1.2.1
Node 1.2 is locally verified, so the conflict manager chooses node 1.2.1.
Refinement or Verification : Verification
No method is applicable, therefore the conflict manager decides to verify node 1.2.1. Node 1.2.1, which is a leaf in the current planning tree, can be verified, this means node 1.2.1 is now verified. Propagation of this proof state leads to a new proof state for 1.2, which was locally verified and is now verified too. The current planning tree can be seen in figure 22.

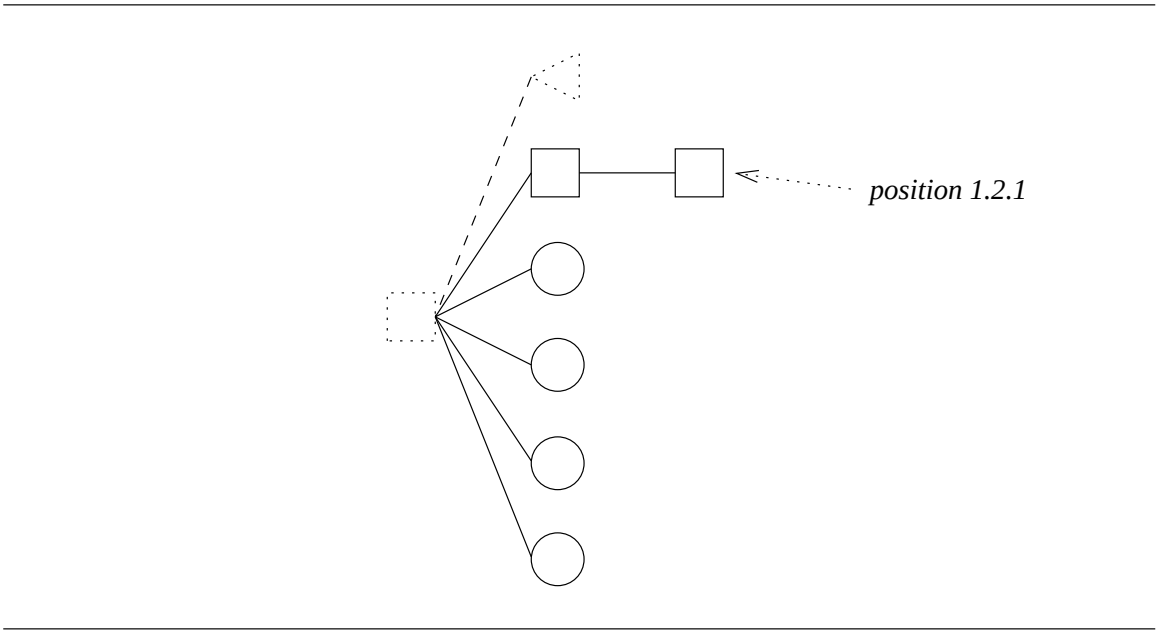

Figure 22: Example 5

Search Node: 1.3
The conflict manager starts to search at node 1.2.1 and goes up to the root, because node 1.2 and 1.2.1 are both verified. The root is locally verified, so the conflict manager chooses node 1.3, on which no proof has been tried.
Refinement or Verification : Refinement
Node 1.3 is open, no proof has been tried and the conflict set is not empty. Hence, a refinement step is chosen.
Conflict Set : Method2
Activate : Method2 creates nodes 1.3.1 and 1.3.2, marks node 1.3 as 'insufficient' and will be inserted in the forbidden-method list of node 1.3.

Search Node: 1.3
Refinement or Verification : Verification
No method is applicable. Node 1.3 can be verified with the help of 1.3.1 and 1.3.2 and is locally verified.

Search Node: 1.3.1
Refinement or Verification : Verification
No method is applicable. Node 1.3.1 can be verified and its proof-state is changed to verified.

Search Node: 1.3.2
Refinement or Verification : Refinement
Conflict Set : Method4
Activate : Method4 creates node 1.3.2.1, marks node 1.3.2 as 'insufficient' and is forbidden on node 1.3.2. The current planning tree can be seen in figure 23.

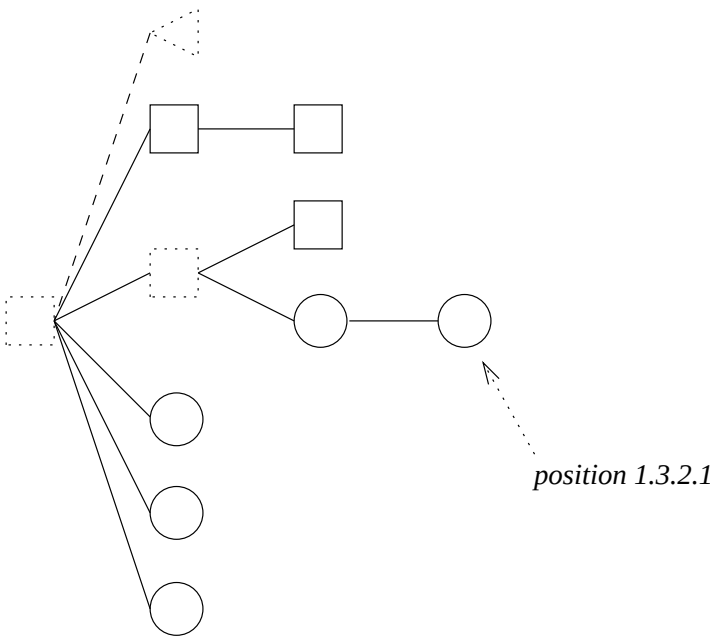

Figure 23: Example 6

Search Node: 1.3.2
Refinement or Verification : Verification
Node 1.3.2 can be verified with node 1.3.2.1 as a used additional lemma and is hence locally verified.

Search Node: 1.3.2.1
Refinement or Verification : Verification
Node 1.3.2.1 can be verified and its proof-state is set to verified. After propagation of this proof state, the nodes 1.3, 1.3.2 are verified, too.

Search Node: 1.4
Refinement or Verification : Refinement
Conflict Set : Method2
Activate : Method2 creates nodes 1.4.1 and 1.4.2 marks node 1.4 as 'insufficient' and is now forbidden on node 1.4.

Search Node: 1.4
Refinement or Verification : Verification

Node 1.4 can be verified with nodes 1.4.1 and 1.4.2 as used additional lemmata and is hence locally verified.

Search Node: 1.4.1
Refinement or Verification : Verification
Node 1.4.1 can be verified and its proof-state is set to verified.

Search Node: 1.4.2
Refinement or Verification : Refinement
Conflict Set : Method4
Activate : Method4 creates node 1.4.2.1, marks node 1.4.2 as 'insufficient' and is now forbidden on node 1.4.2.

Search Node: 1.4.2
Refinement or Verification : Verification
Node 1.4.2 can be proven with node 1.4.2.1 and is, hence, locally verified.

Search Node: 1.4.2.1
Refinement or Verification : Verification
Node 1.4.2.1 can be verified and is, hence, verified. After propagation of this new proof state nodes 1.4 and 1.4.2 are both verified. The current planning tree can be seen in figure 24.

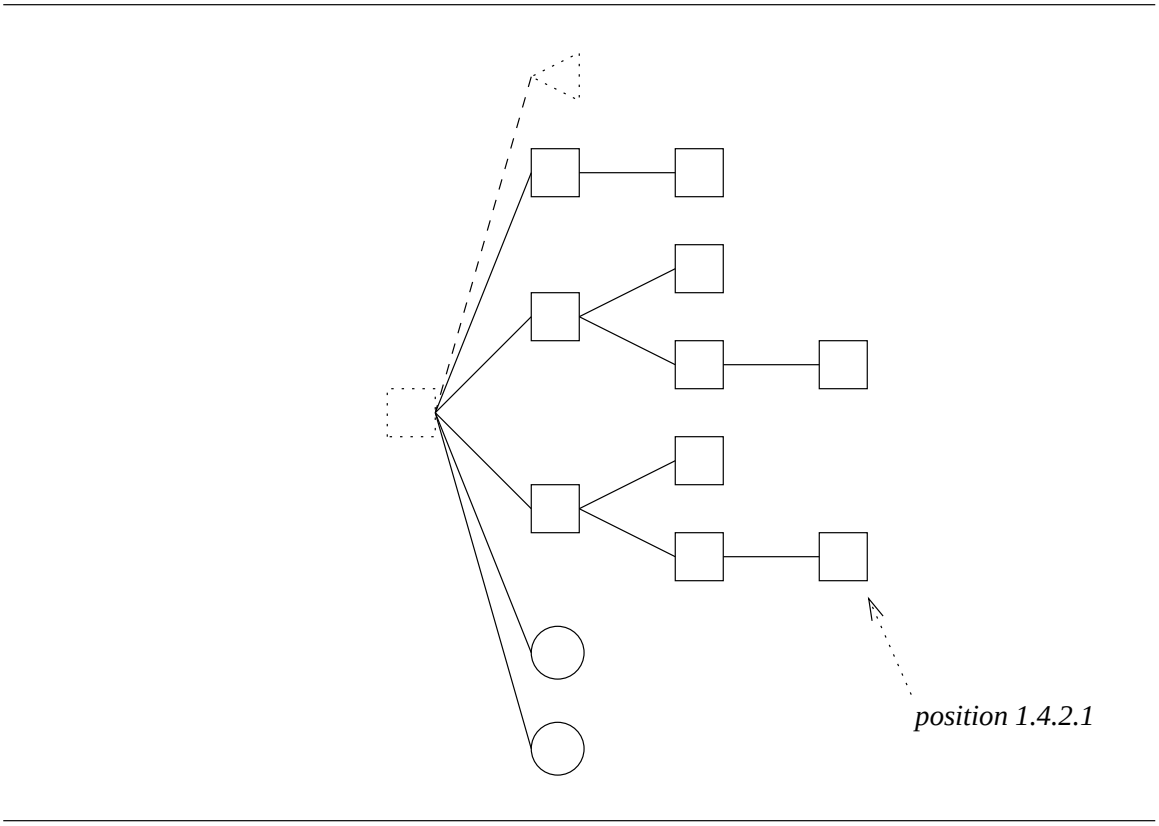

Figure 24: Example 7

Search Node: 1.5
Refinement or Verification : Refinement

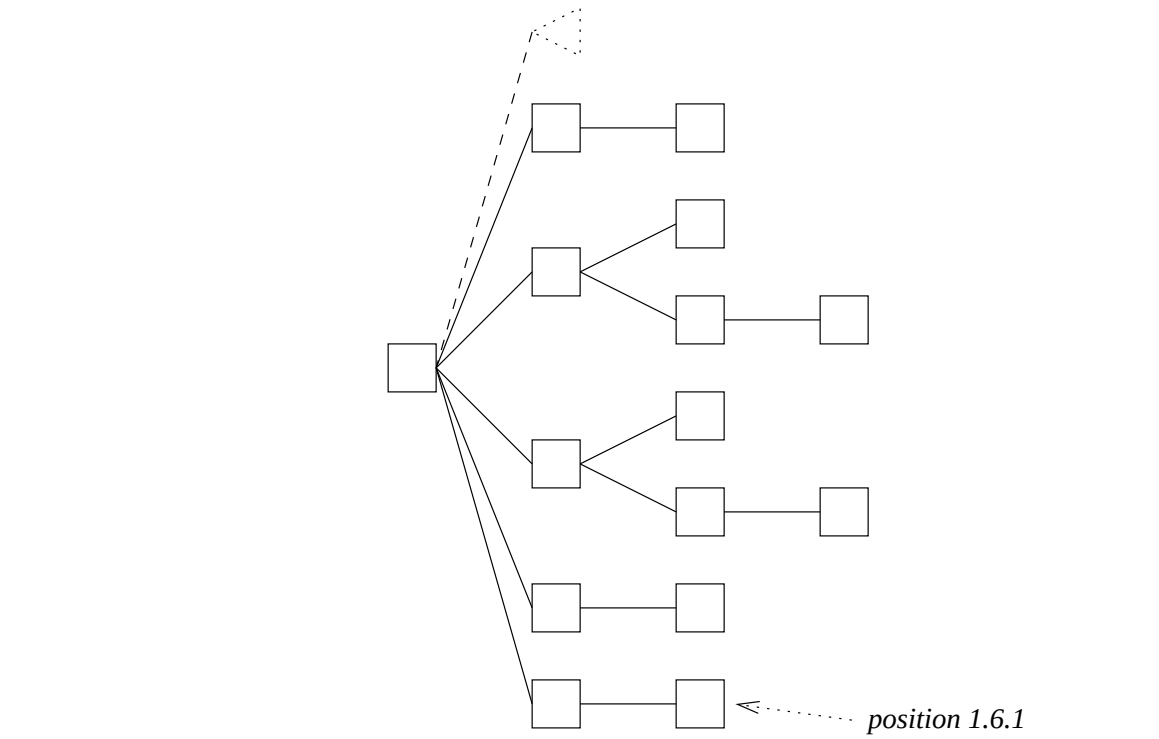

Figure 25: Example 8

Conflict Set : Method4
Activate : Method4 creates node 1.5.1, marks node 1.5 as 'insufficient' and is now forbidden on node 1.5.

Search Node: 1.5
Refinement or Verification : Verification
Node 1.5 can be verified with node 1.5.1 and is, hence, locally verified.

Search Node: 1.5.1
Refinement or Verification : Verification
Node 1.5.1 can be verified and is, hence, verified. After Propagation of this new proof state node 1.5 is verified, too.

Search Node: 1.6
Refinement or Verification : Refinement
Conflict Set : Method4
Activate : Method4 creates node 1.6.1, marks node 1.6 as 'insufficient' and is now forbidden on node 1.6.

Search Node: 1.6
Refinement or Verification : Verification
Node 1.6 can be verified with node 1.6.1 and is, hence, locally verified.

Search Node: 1.6.1
Refinement or Verification : Verification

Node 1.6.1 can be verified and is, hence, verified. After Propagation of this new proof state nodes 1 and 1.6 are both verified. The planning tree is now complete.The main theorem in node 1 has been proven to be a correct theorem. The current planning tree can be seen in figure 25.

8.2 Conjectures

In this section a list of all conjectures occurring in the presented example is given. Every conjecture is associated with the corresponding node by the respective position of the node.

NODE	CONJECTURE
1	`ord(qsort(l)) == true`
1.1	`ord(app(l,cons(n,u))) == and(ord(u),ord(l))`
1.1.1	`<=NL(x,app(l,cons(n,u))) ==` `and(<=NL(x,l),<=NL(x,cons(n,u)))`
1.2	`ord(app(l,cons(x,u))) ==` `and(and(<=LN(l,x),<=NL(x,u)),and(ord(u),ord(l)))`
1.2.1	`<=NL(x,app(l,cons(y,u))) ==` `and(<=NL(x,l),<=NL(x,cons(y,u)))`
1.3	`<=LN(qsort(l),x) == <=LN(l,x)`
1.3.1	`<=LN(app(l,cons(x,u)),y) ==` `and(<=LN(l,y),and(<=(x,y),<=LN(u,y)))`

1.3.2	`and(<=LN(lower(x,l),y),`
	`and(<=(x,y),<=LN(greater(x,l),y))) ==`
	`and(<=LN(l,y),`
	`and(<=(x,y),<=LN(l,y)))`

1.3.2.1	`<=LN(if-list(b,l1,l2),x) ==` `if-bool(b,<=LN(l1,x),<=LN(l2,x))`
1.4	`<=NL(x,qsort(l)) == <=NL(x,l)`
1.4.1	`<=NL(x,app(l,cons(y,u))) ==` `and(<=NL(x,l),and(<=(x,y),<=NL(x,u)))`

1.4.2	`and(<=NL(x,lower(y,l)),`
	`and(<=(x,y),<=NL(x,greater(y,l))))==`
	`and(<=NL(x,l),and(<=(x,y),<=NL(x,l)))`

1.4.2.1	`<=NL(x,if-list(b,l1,l2)) ==` `if-bool(b,<=NL(x,l1),<=NL(x,l2))`
1.5	`<=NL(x,greater(x,l)) == true`
1.5.1	`<=NL(x,if-list(b,l1,l2)) ==` `if-bool(b,<=NL(x,l1),<=NL(x,l2))`
1.6	`<=LN(lower(x,l),x) == true`
1.6.1	`<=LN(if-list(b,l1,l2),x) ==` `if-bool(b,<=LN(l1,x),<=LN(l2,x))`

8.3 User Interaction

The verifier demands two types of user interaction. The first type requires a decision on so called 'inductively complete positions' (refer to [GrLi91] for more information). The latter type expects the user to extend the precedence of the simplification ordering RPOS used by

```
Rule :
======
and(<=NL(N,X4),ORD(X4)) --> ORD(X4)

Inductively complete positions:

**********************
*                    *
*  1 (2): ORD [X4]   *
*  2 (1): <=NL [X4]  *
*                    *
**********************

choose position> 1
```

Figure 26: Inductively Complete Positions

the verifier to orient equations. The required user interaction is illustrated with the help of two examples.

During the verification process of node 1.1 with node 1.1.1 as additional lemma, the verifier determines a set of inductively complete positions, from which one has to be chosen by the user (cf. figure 26).

During the verification process of node 1.3.2.1 the verifier determines a critical pair, which has to be oriented but is incomparable with respect to the simplification ordering RPOS. One way to make an orientation possible is to extend the current precedence. The user input 'g' (stands for 'generate') leads to the generation of a set of nine possible extensions, from which the seventh is strong enough to orient the equation from left to right. This will be indicated by the flag 'straight' (cf. figure 27).

```
Orient uncomparable pair:
        <=LN(IF-LIST(B,L1,L2),X10) ==
        IF-BOOL(B,<=LN(L1,X10),<=LN(L2,X10))


orient pair> g


     ************************************************************
     *                                                          *
     *  1  IF-LIST > <=LN       6  IF-LIST = IF-BOOL            *
     *  2  <=LN > IF-LIST       7  <=LN > IF-BOOL : straight    *
     *  3  IF-LIST = <=LN       8  IF-BOOL > <=LN               *
     *  4  IF-LIST > IF-BOOL    9  <=LN = IF-BOOL               *
     *  5  IF-BOOL > IF-LIST                                    *
     *                                                          *
     ************************************************************


orient pair> 7
```

Figure 27: Orient Uncomparable Pair

9 Conclusions

This chapter summarizes the main results of this paper by first reviewing the work that has been carried out. Secondly, aspects concerning the future prospects of our research are discussed.

9.1 Review

PLATIN has been presented by describing the main components, namely sensors, methods, planning tree, conflict manager and verifier. A detailed example has been given to illustrate the systematization and the interplay of these components. *Sensors* can be seen as predicates on system parameters, returning truth values on these parameters whenever this is required in the planning process. System parameters of interest are, for example, size of the planning tree or structure of the current node. *Methods* provide procedures by which new equations can be generated from an equation which has to be proved. The use of a method is restricted by its preconditions and its conflict information. Preconditions are composed of predicates that evaluate system parameters. The evaluation of system parameters is generally performed by sensors which, in turn, can be employed to specify preconditions. The conflict information allows the evaluation of the possible success of the application of the respective method, depending on the current situation. This allows a comparison between all applicable methods on the basis of their predicted success. Plans are represented by *planning trees*. A node of a planning tree contains relevant information for the planning process like a conjecture and control information. The proof state is an important control information, which describes a relation between the respective node and its parent and allows hypotheses to be handled correctly in the planning and verification process. The heart of the system, the *conflict manager*, can be seen as a control unit, which analyses the plan and determines the course of further actions. In a given situation several methods may be applicable. The conflict manager uses the current sensor values to determine the method preconditions, which ensure the methods adequate application. In case several methods are applicable, the conflict manager orders these methods according to the value of their conflict information and activates the one(s) with the highest value(s). Other important tasks of the conflict manager lie in searching a valuable node to work on and deciding on refinement or verification. The former is currently achieved by a sophisticated depth-first-search strategy. The latter is decided on by evaluating the additional information stored in the planning tree. Every node can contain additional information like 'I may be verifiable' or 'I am insufficient to prove my father', which has been derived from the method responsible for its existence. The *verifier* performs the actual proof steps, it represents a traditional automatic theorem prover. In principal, the verifier is used for two purposes, namely to guarantee the correctness of an unsafe problem decomposition and further to perform basic proof steps, i.e. to prove a leaf [9] of a planning tree.

Generally speaking, the planning approach realized in PLATIN can be seen as situation based planning, which has been extensively studied on many examples like the verification of quicksort (cf. section 8) and other sorting algorithms. As our studies have shown, considerable more theorems can be proved automatically by PLATIN (with UNICOM as underlying verifier) than by UNICOM alone. Furthermore, we believe the open architecture of PLATIN to be especially suited not only for inductive and first order proof planning, but also to increase the flexibility and strength of existing systems.

[9] In this case the verifier can guarantee the correctness or falseness of the appropriate conjecture by proving or rejecting it, respectively.

9.2 Future Lines

The planning approach presented in this paper allows, in our opinion, a wide range of possible applications. For example, new inference rules for a given verifier can be easily integrated and tested in PLATIN. This can be done with the concept of strong methods. A strong method sets the proof state of the corresponding node to 'verified' [10]. This implies that the method itself guarantees the correctness of the problem decomposition. In fact, the concept of strong methods could even replace the underlying verifier. However, due to the current non-hierarchic structure of our sensors and methods this realization is still unfeasible. Furthermore, this sort of structure leads to other questions concerning the relation between conflict information and precondition of methods. A system consisting of methods with 'strong' preconditions often has, during its planning processes, 'small' conflict sets. Hence, it suffices to realize 'simple' conflict information. In an analogous manner 'strong' conflict information leads to 'simple' preconditions. Further research activities have to achieve a better understanding of the relation between precondition and conflict information.

Another topic to be further investigated, is the dynamic concurrency behaviour of methods. In order to coordinate the behaviour of a single method with the behaviour of its concurrent methods, a dynamic and situation suited adjusting of conflict information and preconditions becomes desirable. We believe a hierarchic structure of methods to be a basic step towards such a goal. In addition, a hierarchic structure of sensors could lead to greater efficiency of PLATIN. Note, that currently an evaluation of all sensors is necessary to compute the conflict set, i.e. in every cycle of the conflict manager's main procedure all sensors have to be evaluated at one stage. Furthermore, the question of how the event-driven methods can be compared to the tactics of other planning/proof systems has still to be answered. Naturally, future work will not only include an expansion and refinement of PLATIN, but also research concerning further domains to apply such a system.

[10] Normally this will only be done by the verifier.

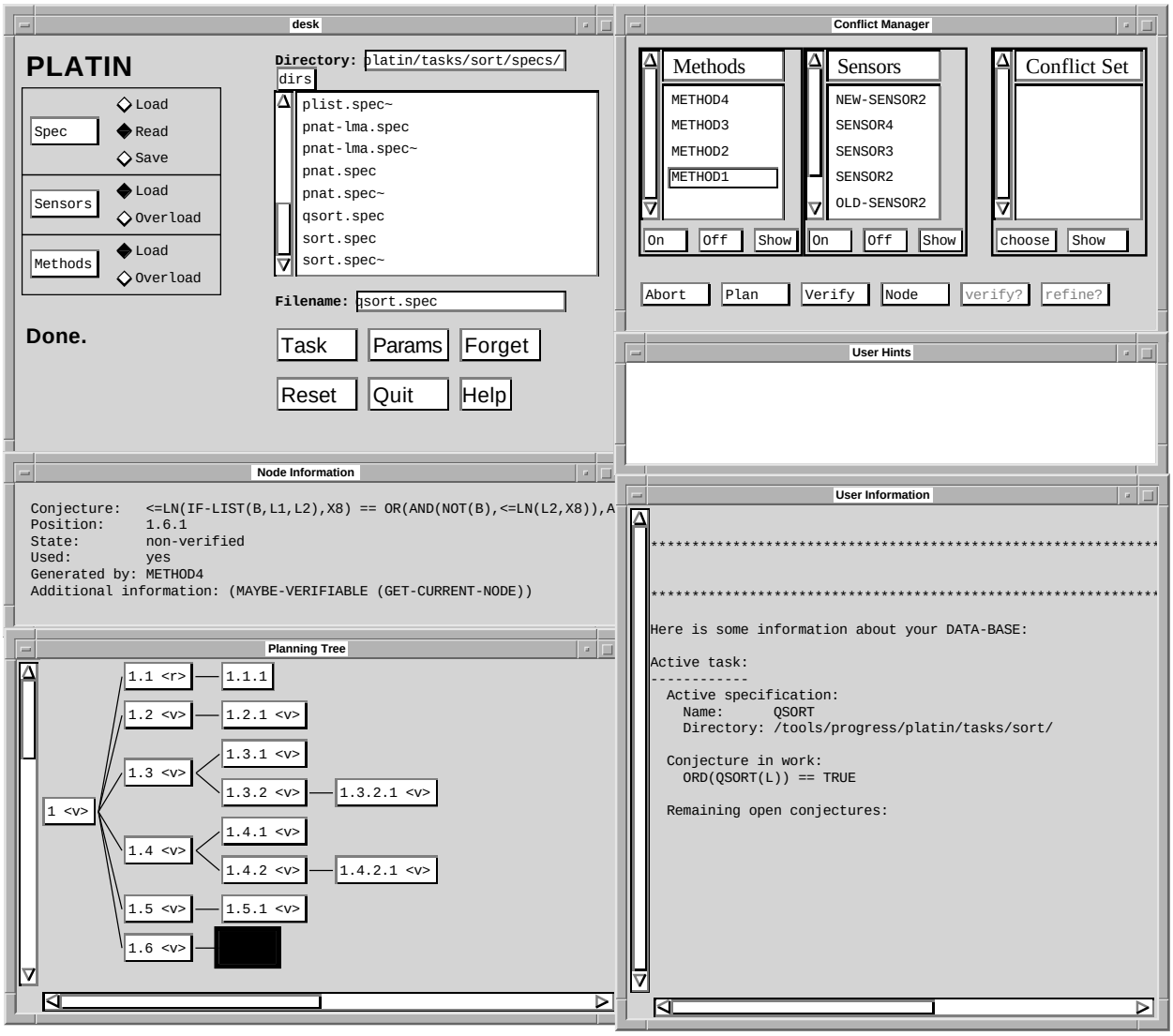

Figure 28: User Interface

A User Interface

To invoke PLATIN the expression '(load-platin)' has to be evaluated in a lisp-environment. The **user interface** of PLATIN consists of six main windows : desk, node information, planning tree, conflict manager, user hints, user information.

Desk is a dialog box featuring a scrolling menu to display the current files. The currently chosen file appears in the filename box. If the current file is a specification, it can be loaded, read or saved. A specification can only be read if it has been loaded once before. Sensor (method) files can be added to the already existing sensors (methods) or replace the already existing sensors (methods). The former is achieved by pressing the load button (diamond shaped), the latter by pressing the overload button followed by the sensor (method) button. By pressing the tree button the current planning tree is saved to a file of the chosen directory, the file name consists of the first part of the name in the filename box followed by the suffix '.tree'. Furthermore, desk allows to leave PLATIN (quit button), to read the help file (help button), to remove the current specifications from PLATIN (forget button) and to remove the current specifications as well as all the loaded methods and sensors from PLATIN (reset button). The two remaining buttons, task and params have accompanying dialog boxes.

The task dialog box is not shown while the one of params is given in figure 29.

Node information shows all information of a node as described in section 5. The node being displayed in this window is the one marked in the *planning tree* window. The latter displays the structure of the current planning tree. At each node a pull-down menu can be activated with the following options: (1) `set-act`: The current node of PLATIN will be set to the highlighted one. (2) `delete`: The highlighted node, including its subtree will be removed from the planning tree. (3) `insert-new`: A new node can be inserted as a direct child of the highlighted node. Note that the highlighted node is the one displayed in the node information while the framed one is the currently chosen one.

Conflict manager consists of three scrolling menus which show the names of the currently loaded sensors and methods as well as the conflict set. Methods and sensors can be switched off and on, more precisely, a method (sensor) can be made into an active or inactive method (sensor) of PLATIN, to either participate or not in the subsequent planning process. All dependencies between methods and sensors will be considered, e.g. switching off a sensor will lead to an inactivation of all methods referring to this sensor. The buttons 'On' and 'Off' are self-explaining. With 'Show' the set of all active and inactive methods (sensors) will be displayed in the corresponding scrolling menu (alternating). During a planning run the conflicting methods appear in the conflict set menu. If, for example the automatic choosing is switched off, all the marked methods will be activated after pressing the choose button. Further the conflict manager window features the following buttons: (1) `Abort`: At any time the user can interrupt any action. (2) `Plan`: To invoke planning this button has to be pressed. (3) `Verify`: The verifier will be started with the conjecture of the selected (highlighted) node. (4) `Node`: The node currently displayed in node information will be selected. (5) `Verify?`: If the user is requested to decide on verification or refinement this button activates the verification. (6) `Refine?`: If the user is requested to decide on verification or refinement this button activates the refinement.

Requests for user interaction will be displayed in the window *user hints*. All information concerning specifications, sensors and methods will be shown in the *user information* window, when requested by the user.

PLATIN's **parameters**, see figure 29, are divided into four groups. The first group, referring to the *Trace* of system information, has only one element, the trace-level. By dragging the slider, the depth of the trace level can be adjusted. The higher the level the more information will be provided. For example, the message that a method is forbidden, will be displayed from level 2 upwards while information concerning a search of a node will be provided from level 5 upwards. The second group *Lemmata* refers to the visibility of lemmata. Thereby a distinction will be made between lemmata generated by methods (regular lemmata) and lemmata produced within a verification process (verifier lemmata). For both sorts of lemmata some kind of visibility parameters can be adjusted, namely scope, classification and default-visibility. A lemma with a global scope can be used within all verification processes of a given specification, whereas a lemma with a task scope can only be used in a proof of a theorem with the same task. The classification of new generated lemmata can be done by either the system or the user. In the first case, the classification will be performed according to the default-visibility parameter. A visible lemma can be seen both inside and outside a specification, while a hidden lemma can only seen inside a specification. The next group, *Conflict Manager Parameters*, is related to the interaction concerning the main procedure (cf. figure 7) and some statistics on used methods. Figure 7 shows the possible interaction between a user and the conflict manager in its main procedure. There are three interaction points (IP's), each of which asks for a decision, which can be given either by the system or by a user. With NEXT-NODE the user can take control of the IP

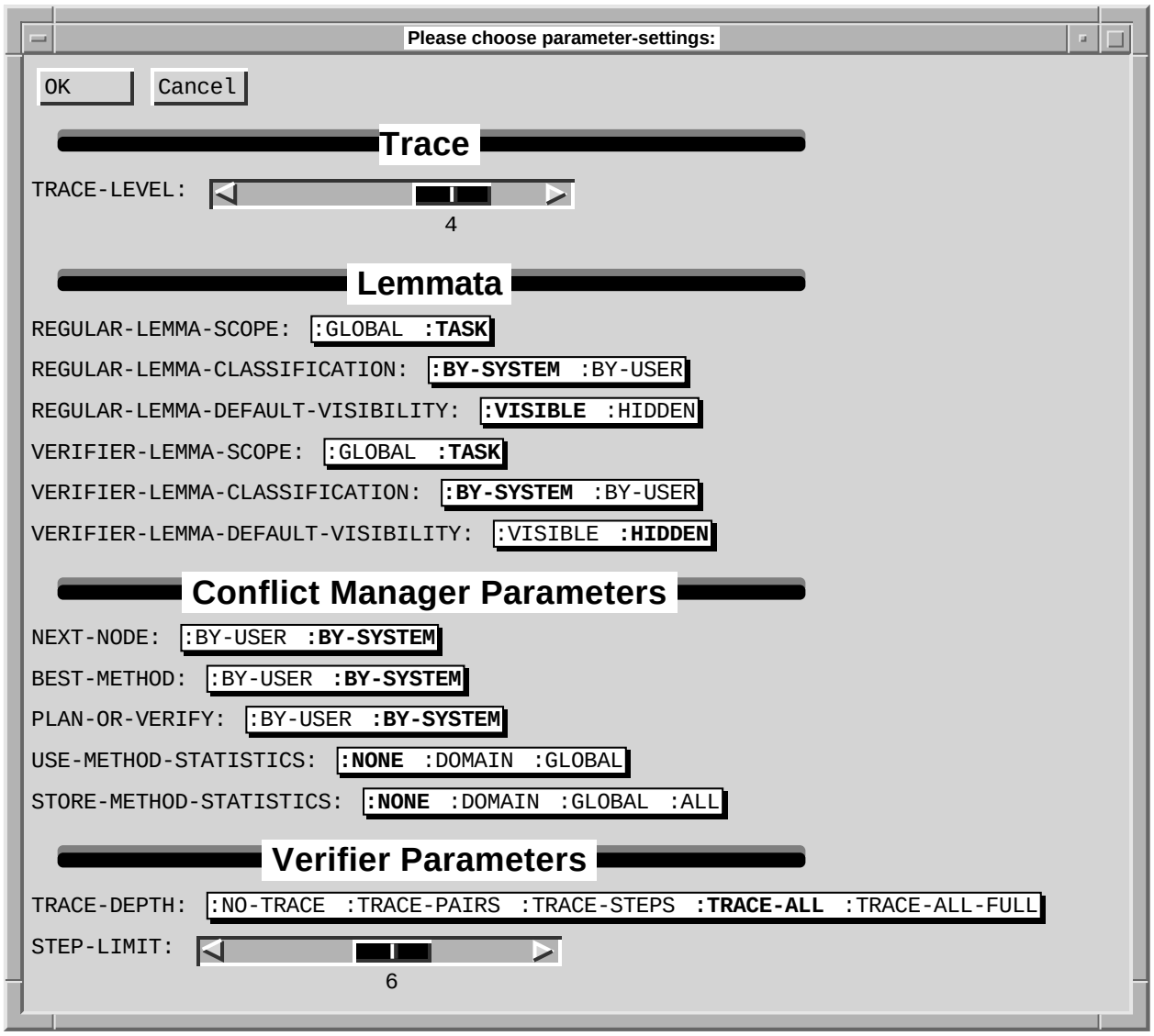

Figure 29: Parameters

search-node, with BEST-METHOD the IP *choose-methods* and with PLAN-OR-VERIFY the IP *refine-current-plan?*. Furthermore, the user can store and use some statistics on used methods. This can be done in a global or a domain specific manner. The last group *Verifier Parameters* comprises two components, namely a property sheet 'TRACE-DEPTH' and a slider 'STEP-LIMIT'. The user can adjust the level of trace information given by the verifier by selecting one element of the property sheet. By dragging the slider the maximal number of verification steps can be set to a number between 1 and 10.

B Example

B.1 Sensor and Method Descriptions

sensor1 (?sort ?symb):
`currentTheoremLhs` contains a subterm of the form `<?symb>(<?sort>(x))` where `?symb` is a defined function symbol of boolean range and arity = 1, `?sort` is some sorting function symbol and x is a variable.

sensor2 (?symb):
`currentTheoremLhs` contains a defined function symbol `?symb`. `?symb` does not appear in `currentTheoremRhs`. `?symb` is not the top-symbol of `currentTheoremLhs`. One of the recursive cases of the definition of `?symb` is an `if-then-else` construction.

sensor3 (?symb ?term):
In `currentTheoremLhs` the function `?symb` is applied to a subterm t of `?term` for which the defined function symbol has an arity greater than 1.

sensor4 (?sort):
The value of `?sort` occurs in `currentTheoremLhs`.

gen-lemmata? (?gen-lemmata):
`current-node` contains some critical pairs, which were generated by some verifier's attempt to prove current-theorem and can probably be used for the purpose of generalization.

method1:
sensors: sensor1
Generation of the left hand side of a new theorem `newTheorem` from the recursive case of `?sort` (generalize each recursive occurrence of `?sort` in the right hand side of the recursive case by a new, distinctive variable).

method2:
sensors: sensor4
Generation of several new theorems. The first, `newTheorem` is generated from the left hand sides of the recursive case of `?sort` with the help of generalization. Further theorems are created by stepwise taking the generalization back.

method3:
sensors: sensor3
Generation of a theorem that shows dependencies of a symbol `?symb`, that the function `ord` is applied to, and the function `<=nl` used in the recursive case of `ord`.

method4:
sensors: sensor2
Generation of the left hand side of a new theorem `newTheorem` from the recursive case of `?inner-symb` by generalizing the recursive occurrence of `?inner-symb` in the right hand side of the recursive case by a new variable; right hand side is generated by taking the right hand side of the recursive case of `?outer-symb`.

vgen-top:
sensors: gen-lemmata?
Generating new theorems by generalizing one critical pair, which was generated by the verifier when proving `currentTheorem`. The critical pair is generalized by removing the top symbols of its left hand side and right hand side, if they are equal, and by then forming new equations from the corresponding arguments.

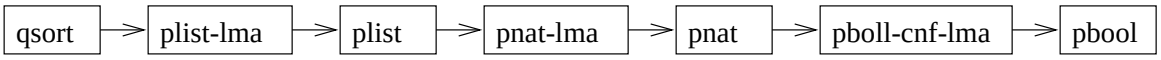

Figure 30: Module Hierarchy of Specification qsort

B.2 Specification qsort

B.2.1 Module qsort

```
SPEC   qsort
USE    plist-lma
SORTS
OPS
   qsort:    list      --> list
   lower:    nat list  --> list
   greater:  nat list  --> list
RULES
   qsort(el)        --> el
   qsort(cons(n,l)) --> app(qsort(lower(n,l)),cons(n,qsort(greater(n,l))))
   lower(n,el)        --> el
   lower(n,cons(m,l)) --> if-list(<=(m,n),
                                  cons(m,lower(n,l)),
                                  lower(n,l))
   greater(n,el)        --> el
   greater(n,cons(m,l)) --> if-list(<=(m,n),
                                    greater(n,l),
                                    cons(m,greater(n,l)))

EQUATIONS
  ord(qsort(l)) = true
ENDSPEC
```

B.2.2 Module plist-lma

```
SPEC   plist-lma
USE    plist
EQUATIONS
   and(<=(n,m),and(<=nl(n,l),<=nl(m,l))) = and(<=(n,m),<=nl(m,l))
   and(not(<=(n,m)),and(<=nl(n,l),<=nl(m,l))) = and(not(<=(n,m)),<=nl(n,l))
ENDSPEC
```

B.2.3 Module plist

```
SPEC   plist
USE    pnat-lma
SORTS  list
OPS
   el:                    --> list
```

```
   cons:    nat list       --> list

   car:     list           --> nat
   cdr:     list           --> list

   if-list: bool list list --> list
   list_=:  list list      --> bool

   <=nl:    nat list       --> bool
   <=ln:    list nat       --> bool

   app:  list list         --> list
   reverse: list           --> list

   ord:   list             --> bool
   noc:   nat list         --> nat
CONSTRUCTORS   el, cons
C-SYMBOLS      list_=
RULES
   car(el)           --> o
   car(cons(n,list)) --> n
   cdr(el)           --> el
   cdr(cons(n,list)) --> list

   if-list(true,l1,l2)  --> l1
   if-list(false,l1,l2) --> l2

   list_=(el,el)                 --> true
   list_=(el,cons(n,list))       --> false
   list_=(cons(n,list),el)       --> false
   list_=(cons(n,l1),cons(m,l2)) --> and(eq(n,m),list_=(l1,l2))

   <=nl(n,el)        --> true
   <=nl(n,cons(m,l)) --> and(<=(n,m),<=nl(n,l))

   <=ln(el,n)        --> true
   <=ln(cons(n,l),m) --> and(<=(n,m),<=ln(l,m))

   app(el,list)       --> list
   app(cons(n,l1),l2) --> cons(n,app(l1,l2))

   reverse(el)           --> el
   reverse(cons(n,list)) --> app(reverse(list),cons(n,el))

   ord(el)           --> true
   ord(cons(n,list)) --> and(<=nl(n,list),ord(list))

   noc(n,el)           --> o
   noc(n,cons(m,list)) --> if-nat(eq(n,m),s(noc(n,list)),noc(n,list))
```

```
EQUATIONS
   reverse(app(x,y))   = app(reverse(y),reverse(x))
   car(if-list(b,x,y)) = if-nat(b,car(x),car(y))
ENDSPEC
```

B.2.4 Module pnat-lma

```
SPEC    pnat-lma
USE     pnat
EQUATIONS
   and(<=(n,m),<=(m,n)) = eq(n,m)
   or(<=(n,m),<=(m,n))  = true
   and(not(<=(n,m)),<=(m,n)) = not(<=(n,m))
   eq(n,n) = true
   <=(n,o) = eq(n,o)
   and(<=(n,m),and(<=(q,m),not(<=(q,n)))) = and(<=(q,m),not(<=(q,n)))

   <=(x,x) = true

   not(<=(x,y)) = and(<=(y,x),not(eq(x,y)))
   and(and(<=(y,x),not(eq(x,y))),or(<=(x,y),b)) = and(not(<=(x,y)),b)
   and(<=(x,y),and(or(not(eq(x,y)),b),or(<=(y,x),b))) = and(<=(x,y),b)
   or(<=(x,y),and(<=(y,x),not(eq(x,y)))) = true
   or(and(<=(x,y),b),and(<=(y,x),and(not(eq(x,y)),b))) = b
   and(or(not(<=(x,m)),not(<=(n,m))),or(<=(x,n),not(<=(n,m)))) =
     not(<=(n,m))
   if-nat(not(<=(n,m)),x,y) = if-nat(<=(n,m),y,x)

   and(or(<=(x,y),b),and(or(<=(y,x),b),or(not(eq(x,y)),b))) = b
   and(<=(x,z),or(<=(x,y),<=(y,z))) = <=(x,z)

   and(<=(x,y),and(<=(y,z),<=(x,z))) = and(<=(x,y),<=(y,z))

   or(<=(x,y),not(eq(x,y))) = true
   or(<=(x,y),eq(x,y)) = <=(x,y)
   or(and(eq(x,y),b),and(<=(x,y),b)) = and(<=(x,y),b)

   and(<=(n,y),or(<=(n,x),eq(x,y))) = and(<=(n,y),<=(n,x))
   and(<=(z,y),or(<=(x,y),not(eq(x,z)))) = <=(z,y)

   and(<=(x,z),and(or(<=(v,z),<=(x,y)),<=(v,y))) = and(<=(x,z),<=(v,y))

   and(or(<=(n,y),<=(x,y)),and(or(<=(n,y),<=(x,z)),or(<=(n,z),<=(x,z)))) =
    and(or(<=(n,y),<=(x,y)),or(<=(n,z),<=(x,z)))
   and(<=(n,x),and(or(<=(n,y),<=(x,y)),or(<=(n,y),not(eq(x,y))))) =
    and(<=(n,x),<=(n,y))
   or(and(eq(x,y),and(<=(n,x),b)),and(<=(n,x),and(<=(n,y),b))) =
    and(<=(n,x),and(<=(n,y),b))
   or(and(<=(u,v),and(<=(z,u),and(not(eq(v,u)),b))),
```

```
      and(<=(z,v),and(<=(v,u),b))) =
    and(<=(z,v),and(<=(z,u),b))
   and(or(<=(x,y),<=(n,y)),or(not(<=(x,y)),<=(n,x))) = and(<=(n,y),<=(n,x))
ENDSPEC
```

B.2.5 Module pnat

```
SPEC  pnat
USE   pbool-cnf-lma
SORTS
   nat
OPS
      o                  : --> nat
      s                  : nat --> nat
      sub,add            : nat nat --> nat
      p                  : nat --> nat
      nat_>,eq           : nat nat --> bool
      <=                 : nat nat --> bool
      if-nat             : bool nat nat --> nat
      max_nat            : nat nat --> nat
CONSTRUCTORS
   o, s
AC-SYMBOLS  add, max_nat
C-SYMBOLS   eq
RULES
    add(o,x) --> x
    add(s(x),y) --> s(add(x,y))

    sub(x,o) --> x
    sub(o,x) --> o
    sub(s(x),s(y)) --> sub(x,y)

    p(o) --> o
    p(s(x)) --> x

    nat_>(o,x) --> false
    nat_>(s(x),o) --> true
    nat_>(s(x),s(y)) --> nat_>(x,y)

    eq(o,o) --> true
    eq(s(x),o) --> false
    eq(o,s(x)) --> false
    eq(s(x),s(y)) --> eq(x,y)

    <=(o,x) --> true
    <=(s(x),o) --> false
    <=(s(x),s(y)) --> <=(x,y)
    if-nat(true,x,y) --> x
    if-nat(false,x,y) --> y
```

```
    max_nat(x,o) --> x
    max_nat(o,x) --> x
    max_nat(s(x),s(y)) --> s(max_nat(x,y))
EQUATIONS
ENDSPEC
```

B.2.6 Module pbool-cnf-lma

```
SPEC  pbool-cnf-lma
USE   pbool
EQUATIONS
  imply(x,y)                      = or(not(x),y)
  if-bool(b,x,y)                  = and(imply(b,x),imply(not(b),y))
  not(not(x))                     = x
  and(x,x)                        = x
  or(x,x)                         = x
  or(x,not(x))                    = true
  and(x,not(x))                   = false
  and(x,or(x,y))                  = x
  and(not(x),or(x,y))             = and(not(x),y)
  and(x,or(not(x),y))             = and(x,y)
  and(or(x,y),or(x,or(not(y),z))) = and(or(x,y),or(x,z))
  and(or(x,y),or(x,not(y)))       = x
  not(and(x,y))                   = or(not(x),not(y))
  not(or(x,y))                    = and(not(x),not(y))
  or(and(x,y),z)                  = and(or(x,z),or(y,z))
ENDSPEC
```

B.2.7 Module pbool

```
SPEC  pbool
SORTS
   bool
OPS
      true,false       : --> bool
      not              : bool --> bool
      and,or           : bool bool --> bool
      imply            : bool bool --> bool
      if-bool          : bool bool bool --> bool
CONSTRUCTORS
   true,false
AC-SYMBOLS and , or
RULES
    not(true) --> false
    not(false) --> true

    and(true,x) --> x
    and(false,x) --> false
```

```
    or(true,x) --> true
    or(false,x) --> x

    imply(true,x) --> x
    imply(false,x) --> true

    if-bool(true,x,y) --> x
    if-bool(false,x,y) --> y
ENDSPEC
```

B.3 Verifier Output

To give an impression of the verifier input/output behaviour, we present the whole verifier input/output occurring during the verification of node 1.1 with node 1.1.1 as used additional lemma.

```
activate verifier
Normalizing essential pair: ORD(APP(X3,CONS(N,X4))) = AND(ORD(X4),ORD(X3))
Result of normalization: ORD(APP(X3,CONS(N,X4))) = AND(ORD(X4),ORD(X3))

Step: 1

Processing pair: ORD(APP(X3,CONS(N,X4))) = AND(ORD(X4),ORD(X3))
of hypothesis  ORD(APP(X3,CONS(N,X4))) = AND(ORD(X4),ORD(X3))


Pair inserted as inductive rule:
  ORD(APP(X3,CONS(N,X4))) --> AND(ORD(X4),ORD(X3))  [qsort 10]


New set of essential critical pairs generated:
     ORD(CONS(N,X4)) = AND(ORD(X4),ORD(EL))
     ORD(CONS(X,APP(L1,CONS(N,X4)))) = AND(ORD(X4),ORD(CONS(X,L1)))

Normalizing essential pair: ORD(CONS(N,X4)) = AND(ORD(X4),ORD(EL))
LHS (0), definition rule: ORD(CONS(N,LIST)) := AND(<=NL(N,LIST),ORD(LIST))
RHS (1), definition rule: ORD(EL) := TRUE
RHS (0), definition rule: AND(TRUE,X) := X
Result of normalization: AND(<=NL(N,X4),ORD(X4)) = ORD(X4)

Normalizing essential pair:
  ORD(CONS(X,APP(L1,CONS(N,X4)))) = AND(ORD(X4),ORD(CONS(X,L1)))
LHS (0), definition rule: ORD(CONS(N,LIST)) := AND(<=NL(N,LIST),ORD(LIST))
RHS (1), definition rule: ORD(CONS(N,LIST)) := AND(<=NL(N,LIST),ORD(LIST))
LHS (1), reduction lemma:
  <=NL(X8,APP(X3,CONS(N,X4))) --> AND(<=NL(X8,X3),<=NL(X8,CONS(N,X4)))
LHS (1), inductive rule: ORD(APP(X3,CONS(N,X4))) --> AND(ORD(X4),ORD(X3))
LHS (2), definition rule: <=NL(N,CONS(M,L)) := AND(<=(N,M),<=NL(N,L))
```

```
Result of normalization:
  AND(AND(<=NL(X,L1),AND(<=(X,N),<=NL(X,X4))),AND(ORD(X4),ORD(L1))) =
  AND(ORD(X4),AND(<=NL(X,L1),ORD(L1)))

End of step: 1
Step: 2

Processing pair: AND(<=NL(N,X4),ORD(X4)) = ORD(X4)
of hypothesis  ORD(APP(X3,CONS(N,X4))) = AND(ORD(X4),ORD(X3))


Pair inserted as inductive rule:
  AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)  [qsort 11]

Normalizing essential pair:
  AND(AND(<=NL(X,L1),AND(<=(X,N),<=NL(X,X4))),AND(ORD(X4),ORD(L1))) =
  AND(ORD(X4),AND(<=NL(X,L1),ORD(L1)))
LHS (0), inductive rule: AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)
RHS (0), inductive rule: AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)
LHS (0), inductive rule: AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)
Result of normalization:
  AND(AND(<=(X,N),ORD(L1)),ORD(X4)) = AND(ORD(X4),ORD(L1))


     Rule :
     ======
     AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)

     Inductively complete positions:

     **********************
     *                    *
     *  1 (2): ORD [X4]   *
     *  2 (1): <=NL [X4]  *
     *                    *
     **********************

choose position> 1

New set of essential critical pairs generated:
     AND(<=NL(N,EL),TRUE) = ORD(EL)
     AND(<=NL(N,CONS(X,LIST)),AND(<=NL(X,LIST),ORD(LIST))) =
       ORD(CONS(X,LIST))

Normalizing essential pair: AND(<=NL(N,EL),TRUE) = ORD(EL)
LHS (1), definition rule: <=NL(N,EL) := TRUE
RHS (0), definition rule: ORD(EL) := TRUE
LHS (0), reduction lemma: AND(X,X) --> X
Result of normalization: TRUE = TRUE
```

```
Normalizing essential pair:
  AND(<=NL(N,CONS(X,LIST)),AND(<=NL(X,LIST),ORD(LIST))) =
  ORD(CONS(X,LIST))
LHS (0), inductive rule: AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)
RHS (0), definition rule:
  ORD(CONS(N,LIST)) := AND(<=NL(N,LIST),ORD(LIST))
LHS (1), definition rule:
  <=NL(N,CONS(M,L)) := AND(<=(N,M),<=NL(N,L))
RHS (0), inductive rule: AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)
LHS (0), inductive rule: AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)
Result of normalization: AND(<=(N,X),ORD(LIST)) = ORD(LIST)

Deleting trivial essential pair: TRUE = TRUE

End of step: 2
Step: 3

Processing pair: AND(<=(N,X),ORD(LIST)) = ORD(LIST)
of hypothesis  ORD(APP(X3,CONS(N,X4))) = AND(ORD(X4),ORD(X3))


Pair inserted as inductive rule:
  AND(<=(N,X),ORD(LIST)) --> ORD(LIST)  [qsort 12]

Normalizing essential pair:
 AND(AND(<=(X,N),ORD(L1)),ORD(X4)) = AND(ORD(X4),ORD(L1))
LHS (0), inductive rule: AND(<=(N,X),ORD(LIST)) --> ORD(LIST)
Result of normalization: AND(ORD(X4),ORD(L1)) = AND(ORD(X4),ORD(L1))

Deleting trivial essential pair:
  AND(ORD(X4),ORD(L1)) = AND(ORD(X4),ORD(L1))


     Rule :
     ======
     AND(<=(N,X),ORD(LIST)) --> ORD(LIST)

     Inductively complete positions:

     **********************
     *                    *
     *  1 (2): ORD [LIST]  *
     *  2 (1): <= [N X]    *
     *  3 (1): <= [X N]    *
     *                    *
     **********************

choose position> 1
```

```
New set of essential critical pairs generated:
     AND(<=(N,X),TRUE) = ORD(EL)
     AND(<=(N,X),AND(<=NL(Y,Z),ORD(Z))) = ORD(CONS(Y,Z))

Normalizing essential pair: AND(<=(N,X),TRUE) = ORD(EL)
LHS (0), definition rule: AND(TRUE,X) := X
RHS (0), definition rule: ORD(EL) := TRUE
Result of normalization: <=(N,X) = TRUE

Normalizing essential pair:
  AND(<=(N,X),AND(<=NL(Y,Z),ORD(Z))) = ORD(CONS(Y,Z))
LHS (0), inductive rule: AND(<=(N,X),ORD(LIST)) --> ORD(LIST)
RHS (0), definition rule:
  ORD(CONS(N,LIST)) := AND(<=NL(N,LIST),ORD(LIST))
LHS (0), inductive rule: AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)
RHS (0), inductive rule: AND(<=NL(N,X4),ORD(X4)) --> ORD(X4)
Result of normalization: ORD(Z) = ORD(Z)

Deleting trivial essential pair: ORD(Z) = ORD(Z)

End of step: 3
Step: 4

Processing pair: <=(N,X) = TRUE
of hypothesis  ORD(APP(X3,CONS(N,X4))) = AND(ORD(X4),ORD(X3))


Pair inserted as inductive rule: <=(N,X) --> TRUE  [qsort 13]


     Rule :
     ======
     <=(N,X) --> TRUE

     Inductively complete positions:

     ********************
     *                  *
     *  1 (): <= [N X]  *
     *  2 (): <= [X N]  *
     *                  *
     ********************

choose position> 1

New set of essential critical pairs generated:
     <=(Z,Y) = TRUE
     TRUE = TRUE
```

```
     EQ(Y,O) = TRUE

Deleting trivial essential pair: TRUE = TRUE

Normalizing essential pair: <=(Z,Y) = TRUE
LHS (O), inductive rule: <=(N,X) --> TRUE
Result of normalization: TRUE = TRUE

Normalizing essential pair: EQ(Y,O) = TRUE
Result of normalization: EQ(Y,O) = TRUE

Deleting trivial essential pair: TRUE = TRUE

End of step: 4
Step: 5

Processing pair: EQ(Y,O) = TRUE
of hypothesis  ORD(APP(X3,CONS(N,X4))) = AND(ORD(X4),ORD(X3))


Pair inserted as inductive rule: EQ(Y,O) --> TRUE  [qsort 14]


New set of essential critical pairs generated:
     TRUE = TRUE
     FALSE = TRUE

Deleting trivial essential pair: TRUE = TRUE

Normalizing essential pair: FALSE = TRUE
Result of normalization: FALSE = TRUE



Hypothesis rejected:
     ORD(APP(X3,CONS(N,X4))) = AND(ORD(X4),ORD(X3))

Inconsistent equation(s):
     FALSE = TRUE

End of step: 5



*****************************

         REJECTED !!

*****************************
```

C Functions

fs-sort: $\langle fsymb \rangle \rightarrow \langle sort \rangle$
Returns the sort of the result of the function $\langle fsymb \rangle$

fs-contain: $\langle fsymb \rangle\ \langle term \rangle \rightarrow$ {true, false}
$true$, if the function $\langle fsymb \rangle$ occurs in $\langle term \rangle$
$false$, otherwise

fs-defined: $\langle fsymb \rangle \rightarrow$ {true, false}
$true$, if $\langle fsymb \rangle$ is a defining function
$false$, otherwise

fs-top: $\langle term \rangle \rightarrow \langle fsymb \rangle$
Returns the leading (outermost) function symbol

fs-rec-rhs: $\langle fsymb \rangle \rightarrow \langle term \rangle$
Returns the right hand side of the defining equation of the function $\langle fsymb \rangle$, that contains a recursive call of $\langle fsymb \rangle$

fs-arity: $\langle fsymb \rangle \rightarrow \mathbf{N}$
Returns the arity of the function $\langle fsymb \rangle$

fs-occur: $\langle fsymb \rangle\ \langle term \rangle \rightarrow \mathbf{N}$
Returns the number of occurrences of $\langle fsymb \rangle$ in $\langle term \rangle$

fs-subterm: $\langle term_1 \rangle\ \langle fsymb \rangle \rightarrow \langle term_2 \rangle$
Returns a subterm of $\langle term_1 \rangle$ with the leading function $\langle fsymb \rangle$

fs-superterm: $\langle term_1 \rangle\ \langle fsymb \rangle \rightarrow \langle term_2 \rangle$
Returns a subterm of $\langle term_1 \rangle$ of which a direct subterm possesses the leading function symbol $\langle fsymb \rangle$

fs-replace: $\langle term_1 \rangle\ \langle fsymb \rangle\ \langle term_2 \rangle \rightarrow \langle term_3 \rangle$
One subterm of $\langle term_1 \rangle$ with the leading function $\langle fsymb \rangle$ is replaced by $\langle term_2 \rangle$

fs-replace-all: $\langle term_1 \rangle\ \langle fsymb \rangle\ \langle term_2 \rangle \rightarrow \langle term_3 \rangle$
All subterms of $\langle term_1 \rangle$ with the leading function $\langle fsymb \rangle$ are replaced by $\langle term_2 \rangle$

varp: $\langle term \rangle \rightarrow$ {true, false}
$true$, if $\langle term \rangle$ is a variable
$false$, otherwise

var-glob-setp: $\langle fsymb \rangle \rightarrow$ {true, false}
$true$, if $\langle fsymb \rangle$ is a bound global variable
$false$, otherwise

var-gen: $\rightarrow \langle term \rangle$
Generates a new variable of appropriate sort

var-extract: $\langle term_1 \rangle \rightarrow \langle term_2 \rangle$
Returns the least nested variable of $\langle term_1 \rangle$ that has the same sort as the term itself

var-con-onep: $\langle term \rangle \rightarrow$ {true, false}
$true$, if each subterm of $\langle term \rangle$ shares at most one variable with any other subterm of $\langle term \rangle$
$false$, otherwise

var-connect: $\langle term_1 \rangle \ \langle term - list \rangle \ \langle fsymb - list \rangle \rightarrow \langle term_2 \rangle$
Returns a term of the form $and(and_i^k(\leq_i \ (x_i, n)), and_j^m(\leq_j \ (y_j, n)))$; n is the least nested variable of $\langle term_1 \rangle$ that does not occur in $\langle term - list \rangle$; the x_i describe the (differing) variables occurring in the flattened form of $\langle term_1 \rangle$ before n; the y_j describe the (differing) variables occurring in flattened form of $\langle term_1 \rangle$ before n; the $\leq_i$ and $\leq_j$ stand for the appropriate (sort considering) functions (all of which have two arguments) from $\langle fsymb - list \rangle$

dir-subterms: $\langle term \rangle \rightarrow \langle term - list \rangle$
Returns a list of all direct subterms of $\langle term \rangle$

subterms: $\langle term \rangle \rightarrow \langle term - list \rangle$
Returns a list of all subterms of $\langle term \rangle$

te-var: $\langle term \rangle \rightarrow \langle term - list \rangle$
Returns a list of all variables of $\langle term \rangle$

te-equalp: $\langle term_1 \rangle \ \langle term_2 \rangle \rightarrow$ {true, false}
$true$, if $\langle term_1 \rangle$ and $\langle term_2 \rangle$ are equal except for variable-names
$false$, otherwise

te-replace: $\langle term_1 \rangle \ \langle term_2 \rangle \ \langle term_3 \rangle \rightarrow \langle term_3 \rangle$

The subterm $\langle term_2 \rangle$ of $\langle term_1 \rangle$ is replaced by $\langle term_3 \rangle$

nd-new: $\langle equation \rangle$ {n, v, u, r} $\langle name \rangle$ $\langle info \rangle \rightarrow \langle node \rangle$
Returns a node of a planning tree

nd-attach: $\langle node \rangle$
Attaches $\langle node \rangle$ as a child to *currentNode* of the current planning tree

List of Figures

1 The Architecture Underlying PLATIN 3
2 Node Structure 12
3 Representation of planning trees 13
4 Propagation I 14
5 Propagation II 15
6 Conflict Manager 16
7 Main Procedure of the Conflict Manager 17
8 Strategies 18
9 Choose Node 19
10 Avoiding unused Subtrees 19
11 Avoiding locally rejected or rejected Subtrees 20
12 Infinite Search Loop 21
13 Decision Tree I 22
14 Decision Tree II 22
15 Decision Tree III 23
16 Verifier 24
17 Example 25
18 Example 1 27
19 Example 2 28
20 Example 3 28
21 Example 4 29
22 Example 5 30
23 Example 6 31
24 Example 7 32
25 Example 8 33
26 Inductively Complete Positions 35
27 Orient Uncomparable Pair 36
28 User Interface 39
29 Parameters 41
30 Module Hierarchy of Specification qsort 43

References

[Av90] Jürgen Avenhaus. *Reduktionssysteme 2* Vorlesungsskript, Fachbereich Informatik, Universität Kaiserslautern, 1990.

[Av91] Jürgen Avenhaus. *Reduktionssysteme 1* Vorlesungsskript, Fachbereich Informatik, Universität Kaiserslautern, 1991.

[Av95] Jürgen Avenhaus. *Reduktionssysteme* Springer-Verlag 1995, ISBN 3-540-58559-1.

[Ba87] L. Bachmair. *Proof Methods for Equational Theories* PhD thesis, Dept. of Computer Science, University of Illinois, Urbana, 1987.

[Ba88] L. Bachmair. *Proof by consistency in equational theories* Proc. of LICS, pages 228–233, 1988.

[BaZa92] James D. Baker and Shahriar Zand-Biglari. *An Integral Theorem Prover and the Role of Proof Planning* Journal of Automated Reasoning, vol. 8, pages 275-295, 1992.

[BoMo79] Robert S. Boyer and J. Strother Moore. *A Computational Logic* Academic Press, 1979.

[BoMo88] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook* Academic Press, 1988.

[Bu88] Alan Bundy. *The use of explicit plans to guide inductive proofs* Proc. of 9th International Conference on Automated Deduction, 1988.

[ChGr91] J. Christensen and A. Grove. *A Formal Model for Classical Planning* Proc. IJCAI-91, vol. 1, ed. J. Mylopoulos and R. Richter, pages 246-251, 1991.

[DeKr94] Jörg Denzinger and Martin Kronenburg. *Planning for distributed theorem proving: The team work approach* SEKI-Report SR-94-09, Fachbereich Informatik, Universität Kaiserslautern, 1994.

[DeMu89] Jörg Denzinger and Jürgen Müller. *Eqtheopogles - a completion theorem prover for pl1eq* In D. Metzing, editor, GWAI-89, 13th German Workshop on Artificial Intelligence, Vol. 216 of Informatik-Fachberichte , pp 92-101, Springer-Verlag, 1989.

[EFT92] H.-D. Ebbinghaus, J. Flum, W. Thomas (1992) *Einführung in die mathematische Logik* BI-Wiss.-Verl., ISBN 3-411-15603-1, 1992.

[EJRS94] Robert Eschbach, Wolfgang Jekeli, Carlo Reiffers and Inger Sonntag. *PLATIN - Implementation und Heuristiken* Fachbereich Informatik, Universität Kaiserslautern, yet unpublished, 1994.

[GrLi91] Bernhard Gramlich and Wolfgang Lindner. *A guide to unicom, an inductive theorem prover based on rewriting and completion techniques* SEKI-Report SR-91-17, Fachbereich Informatik, Universität Kaiserslautern, 1991.

[Gr90] Bernhard Gramlich. *Completion based inductive theorem proving: A case study in verifiying sorting algorithms* SEKI-Report SR-90-04, Fachbereich Informatik, Universität Kaiserslautern, 1990.

[Gr89] Bernhard Gramlich. *Inductive theorem proving using refined unfailing completion techniques* SEKI-Report SR-89-14, Fachbereich Informatik, Universität Kaiserslautern, 1989.

[HHRS86] R. Hähnle, M. Heisel, W. Reif, and W. Stephan. *An interactive verification system based on dynamic logic* In Proc. 8th CADE, LNCS 230, pp 306-315. Springer, 1986.

[HKRS95] Xiaorong Huang and Manfred Kerber and Jörn Richts and Arthur Sehn. *Planning Mathematical Proofs with Methods* Journal of Information Processing and Cybernetics, vol. 30, pages 277-291, 1995.

[HKK92] Xiarong Huang, Manfred Kerber, and Michael Kohlhase. *Methods - the basic units for planning and verifying proofs* Submitted to IJCAI93, 1992.

[My et al.91] Brad A Myers, Dario A. Guise, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish and Philippe Marchal. *Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces* IEEE Computer, vol. 23, no. 522, pp. 187-205, 1991.

[Ne81] A. Newell. *The heuristic of george polya and its relation to artificial intelligence* Technical Report CMU-CS-81-133, Carnegie-Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania, USA, 1981.

[SoDe93] Inger Sonntag and Jörg Denzinger. *Extending automatic theorem proving by planning* SEKI-Report SR-93-02, Fachbereich Informatik, Universität Kaiserslautern, 1993.

Index

Symbols

$sem(\Phi)$ 24
$\vdash_V$ 25

A

additional-information 12

C

conflict information 8
conflict information
    planning 9
    refinement 9
    replanning 9
conjecture 12

D

decomposition 8
deduction 8
deduction theorem 24
defun-sensor 6
derivation 8
description 5

F

forbidden-methods 12

G

generated-by 12

I

inductively complete position 34
IP 40

M

method 8
    conflict information 8
    static 8
    strong 9
    description 8
    dynamic 8
    name 8
    precondition 8
    program 8
    structure 8

N

node 12
    structure 12

O

orientation 34

P

planning tree 12
    consistent 15
postcondition 8
    insufficient 9
    maybe-verifiable 9
    verifiable 9
precondition 8
program 5, 8
proof-state 12
    verified 12
    locally verified 12
    locally rejected 12
    open 12
    rejected 12
proof-tried 12
propagation 14

R

representation 14

S

semantic 24
sensor 5
    -variables 5
    generation of 6
    structure 5
step-limit 26

U

UNICOM 2
used 12
user interaction 34

V

verifier 24
    abstract 24
verifier-steps 26