SETHEO V 3.3

Reference Manual

— Draft —

Johann Schumann Ortrun Ibens

1.1.95,21.4.95,31.7.96

Preface

This is the SETHEO Reference Manual ...

Short history of SETHEO,

Overview of the Manual

`Johann`

Copyright and SETHEO Licence

von den alten Unterlagen

Contents

1 What's New **3**

2 Notation **4**

3 Installing SETHEO **5**

3.1 Introduction 5

3.2 Getting the Binaries 5

3.3 Installing the Binaries 6

4 Getting Started **7**

4.1 Tutorial 1 7

4.1.1 Preparing the formula 8

4.1.2 Running SETHEO 10

4.1.3 The Model Elimination Calculus 12

4.1.4 Looking at the Proof 13

4.2 Tutorial 2 13

4.3 Static Refinements 14

4.3.1 Regularity 14

4.3.2 Tautology and Subsumption Constraints 16

4.3.3 Deletion of Links 17

4.4 Dynamic Constraints: Antilemmata 18

4.5 Additional Inference Rules: Folding Up and Folding Down 19

4.6 Reordering 20

4.6.1 Clause Reordering 20

4.6.2 Subgoal Reordering 21

4.6.3 Subgoal Alternation 23

4.7 Bounds 25

4.7.1 Locality and Globality of Completeness Bounds 26

4.7.2 Completeness Bounds 27
4.8 Exercises 28

5 The SETHEO System **29**
5.1 SETHEO System Overview 29
5.2 Filename Conventions 31
5.3 The Environment 32
5.4 The Basic Programs 32
5.4.1 plop 32
5.4.2 inwasm 33
5.4.3 wasm 36
5.4.4 sam 36
5.4.5 xptree 45
5.5 Additional Modules 47
5.5.1 clop 47
5.5.2 setheo 48
5.5.3 delta 48
5.5.4 xvtheo 50
5.5.5 xvdelta 50

6 File Formats **51**
6.1 The First Order Predicate Logic Syntax 51
6.2 The LOP-Syntax 52
6.3 SAM Assembler Syntax 57
6.3.1 Directives 57
6.3.2 Statements 59
6.4 SAM Machine Code Syntax 59
6.5 Syntax Definition of a Proof Tree 60
6.5.1 The Tree File 60
6.5.2 The Operator Translation Table 62
6.6 The SAM Logfile 63

7 Logic Programming **69**

7.1 Global Variables 69

7.1.1 Syntax and Usage of Global Variables 70

7.1.2 Examples of the Usage of Global Variables 70

7.1.3 Restictions for Global Variables 71

7.2 Built-ins for Test, Arithmetic and Assignment 71

7.2.1 Destructive Assignment, := 71

7.2.2 $unify/2,= 72

7.2.3 $isunifiable/2 72

7.2.4 $isnotunifiable/2 73

7.2.5 $eq/2, == 73

7.2.6 $neq/2, =/= 74

7.2.7 is 74

7.2.8 Relational Operators, $<, >, \geq$ 75

7.2.9 $isvar/1 75

7.2.10 $isnonvar/1 76

7.2.11 $isconst/1 76

7.2.12 $iscompl/1 77

7.2.13 $isnumber/1 77

7.3 Built-ins for Control 78

7.3.1 $eqpred/1 78

7.3.2 $fail/0 78

7.3.3 $cut/0,$precut/0 79

7.3.4 $stop/0 79

7.3.5 $monitor/0 80

7.4 Built-ins for Input-Output 80

7.4.1 $ptree/0 80

7.4.2 $tell/1 81

7.4.3 $told/0 81

7.4.4 $write/1 82

7.5 Built-ins for Bounds and Sizes 82

7.5.1 $get*bound*/1 . . . 82
7.5.2 $set*bound*/1 . . . 83
7.5.3 $size/2 . . . 84
7.5.4 $tdepth/2 . . . 84
7.6 Built-ins for Lemmata . . . 85
7.6.1 $genlemma/2 . . . 85
7.6.2 $genulemma/3 . . . 86
7.6.3 $uselemma/1 . . . 87
7.6.4 $dumplemma/0 . . . 88
7.6.5 $getnrlemmata/1 . . . 89
7.6.6 $delrange/2 . . . 89
7.7 Built-ins for controlling Counters . . . 90
7.7.1 $initcounters/1 . . . 90
7.7.2 $setcounter/2 . . . 90
7.7.3 $getcounter/2 . . . 91
7.8 Built-ins for Processing Terms . . . 91
7.8.1 $functor/3 . . . 91
7.8.2 $arg/3 . . . 92

8 How To . . . 93
8.0.3 Reporting SETHEO Errors . . . 93
8.1 Set Parameters . . . 94
8.2 Selection of Start Clauses . . . 95
8.2.1 Disabling a Query . . . 96
8.2.2 Converting a Clause into a Query . . . 96
8.3 Assembly and Display of Answer Substitutions . . . 96
8.3.1 The Horn Case . . . 96
8.3.2 The Non-Horn Case . . . 97
8.4 Iterative Deepening . . . 98
8.5 Generating and Using Unit-lemmata . . . 98
8.6 Defining New Built-Ins . . . 98

9 What If ... 99

9.1 SAM Error Messages 99

9.2 Return values of the SETHEO system 101

10 Installing the Sources 103

10.1 Introduction 103

10.2 Getting the Sources 103

10.3 Installing the Sources 103

10.4 Migration to New Platforms 103

11 Glossary 107

12 Index 109

1. What's New

2. Notation

`similar to UNIX-manual`

Definition of BNF and regular expressions,

what is **boldface**, *italic*, `type-writer-style`

`johann`

3. Installing SETHEO

3.1 Introduction

This section is about how to install the SETHEO system binaries at your own computer. For sevaral platforms (including Sun Os, Sun Solaris, HPUX, Linux) binaries of the SETHEO code are available via ftp. In addition to the programs examples and manpages are provided.

If you run another operation system or if you want to look at the source code, too, you can also get the sources. Read Chapter 10, how to obtain the sources.

3.2 Getting the Binaries

The SETHEO system can be obtained via anonymous ftp. The ftp–server provides a package containing the binaries of the basic SETHEO programs for specific platforms (see Figure 3.1), the manpages, example problems and a file with installation hints. Binaries for other platforms will be available in the future.

System	Name of Package
Sun Os 4.1.3	setheo.sunos.tar.gz
Sun Solaris 2.4 (or up)	setheo.solaris.tar.gz
HP HPUX	setheo.hp.tar.gz
Linux	setheo.linux.tar.gz

Figure 3.1: Platforms for which binaries of the SETHEO system are provided.

To get this package you have to connect the ftp–server either by

```
ftp ftp.informatik.tu-muenchen.de
```

or by

```
ftp 131.159.0.198
```

If the connection is established, the ftp–server will ask for your name and your e–mail address:

```
Name: anonymous
Password: <your-e-mail-address>
```

The next thing you have to do is change to the right directory, set the transfer mode to `binary` and get the `setheo.solaris.tar.gz` file (respectively `setheo.hp.tar.gz` or `setheo.sunos.tar.gz`, or `setheo.linux.tar.gz`):

```
ftp> cd /local/lehrstuhl/jessen/Automated_Reasoning/SETHEO
ftp> binary
ftp> get setheo.solaris.tar.gz
```

You can get a short version of this description as well. After this leave the ftp–server:

```
ftp> get README
ftp> bye
```

3.3 Installing the Binaries

Now you have a local file `setheo.solaris.tar.gz` (respectively `setheo.hp.tar.gz`, `setheo.sunos.tar.gz` or `setheo.linux.tar.gz`). This is a compressed package containing directories and files of the SETHEO system. Unpack the system with:

```
gunzip -c setheo.tar.gz
tar xvf setheo.solaris.tar
```

You can delete the file `setheo.solaris.tar`, if you want. Go into the directory `setheo.solaris` and call the installation process:

```
cd setheo.solaris
make
```

If you want to invoke SETHEO from any directory, you must add this directory to your path. Now SETHEO is ready for use.

4. Getting Started

After you have fetched and installed SETHEO on your system, you are ready to go. In this chapter we present three tutorials which shall enable you to successfully use the SETHEO system.

The tutorials are suited for different levels of experience in automated theorem proving and thus have different prerequisits.

The first tutorial starts with a problem given in mathematical notation. In a step by step fashion all actions necessary from entering the formula to understanding SETHEO's proof will be described. The only prerequisits needed to understand this tutorial is a general knowledge of logic and mathematics. This tutorial also contains a short informal introduction into the Model Elimination calculus.

The second tutorial aims at readers who are somewhat familiar with the Model Elimination calculus and the basics of automated theorem proving. Given a formula in first order logic (or a set of clauses), the user is guided to explore the different possibilities to set parameters and watch the influence the setting has on the search for the proof.

The third tutorial assumes familiarity with the basics of SETHEO and logic. It describes the steps which are necessary to go from given proof tasks from an application towards the usage of SETHEO. For each step practical hints will be given and problems which might (or will) occur will be discussed.

A number of exercises concludes this chapter.

4.1 Tutorial 1

Assume you have to solve the following problem which is called *non-obviousness problem*[1]:

> Let p and q be two binary relations with the following properties: p and q are transitive. Furthermore, q is symmetric. Additionally, we know that for each pair of elements, the pair is either in relation p or in relation q.
>
> Now we have to show that at least one of the relations p or q is total.

As a first step we have to write down this problem in a formal notation, expanding the definitions of "transitive", "symmetric" and "total". A relation[2] r is transitive, if

$$\forall X, Y, Z \quad XrY \wedge YrZ \rightarrow XrZ,$$

[1] The problem has been taken from [?]. A formulation of that problem is also present in the TPTP [Sutcliffe *et al.*, 1994] as problem `MSC006-1`.

[2] With XrY we denote that X is in relation r with Y.

it is symmetric, if

$$\forall X, Y \quad XrY \rightarrow YrX$$

and total (i.e., all elements are in the relation r), if

$$\forall X, Y \quad XrY.$$

Now, given the formal definitions of all properties, you are invited to take a sheet of paper and try to prove the theorem.

Did you make it? If not, don't be desparate. There is a proof in Table 4.2 at the end of this tutorial and furthermore, there is SETHEO to support you.

In order to use SETHEO to automatically find a proof, you have to write down the problem in first order predicate logic (FOL). In general, such a formula is of the form

$$\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_1 \rightarrow \mathcal{T}$$

where $\mathcal{A}_i$ are Axioms (in our case, the properties of the relations p and q) and $\mathcal{T}$ is the theorem we want to show. When we insert the definitions, we obtain:

$$\begin{array}{lll}
((\forall X, Y, Z & XpY \wedge YpZ \rightarrow XpZ) & \wedge \\
(\forall X, Y, Z & XqY \wedge YqZ \rightarrow XqZ) & \wedge \\
(\forall X, Y & XqY \rightarrow YqX) & \wedge \\
(\forall X, Y & XpY \vee XqY)) & \\
\rightarrow & & \\
((\forall X, Y & XpY) & \vee \\
(\forall X, Y & XqY)) &
\end{array}$$

Before starting SETHEO, however, two further steps must be done: (1) the syntax must be changed and (2) the formula must be converted into Conjunctive Normal Form (CNF). In this tutorial we will only describe, how the transformation is done automatically, using a module of the SETHEO System. The manual transformation of an arbitrary formula into CNF is described in detail in many textbooks. Here, we refer to [Loveland, 1978; ?]. The transformation consists of essentially two steps: removing all logical operators which do not belong to CNF and elimination of the quantifiers by skolemization.

4.1.1 Preparing the formula

Since SETHEO does not understand LaTeX and infix mathematical notation, we have to modify the syntax according to the following rules.

- SETHEO cannot understand infix notation (well, there are some exceptions). Therefore, instead of XrY, we have to write a binary relation r as a predicate of arity two: `r(X,Y)`[3]. This predicate has the meaning: `r(X,Y)` $\equiv \mathbf{T}$ iff X and Y are in the relation r.

[3]Actually, there is quite a number of different ways to convert the syntax. E.g., one could have written instead: `isInRelation(r,X,Y)`. For further details see our third tutorial.

- All variables must start with an uppercase letter or an underscore, e.g., `X, Y, _99, _j`; all predicate-, function- and constant symbols must start with a lowercase letter (e.g., `r, f(X)`).
- The connectives and quantifiers must be changed into an ASCII representation (e.g., $\forall$ must be written as `forall`).

For details on the FOL syntax of SETHEO see Chapter 6. If you choose to do the conversion of a formula into CNF by hand, all steps of transforming the syntax execpt the last one apply.

After applying all syntactic sugar to our formula, we obtain:

```
(
  forall X,Y,Z (p(X,Y) & p(Y,Z) -> p(Y,Z)) &
  forall X,Y,Z (q(X,Y) & q(Y,Z) -> q(Y,Z)) &
  forall X,Y   (q(X,Y) -> q(Y,X)) &
  forall X,Y   (p(X,Y) ;  q(X,Y))
) -> (
  ( forall X,Y  p(X,Y)) ;
  ( forall X,Y  q(X,Y))
)
```

This formula must be kept in a file with the extension **.pl1** (for 1st order predicate logic). Our example is contained in file *MSC006-1.pl1*[4]. Now, we are ready to start the SETHEO system. In a first step, we have to convert this formula into CNF (i.e., a set of clauses). This is done using the module **plop** by typing:

```
plop -neg MSC006-1
```

The `-neg` option must be given, because SETHEO's proof mode is a *refutation* procedure. Instead of proving our theorem, SETHEO tries to refute its negation..

After invoking **plop**, a lot of data are printed on the screen, until finally, the message

```
Generated clauses:  6
```

appears. Above this message, these clauses are printed on the screen. Additionally, all clauses have been written into the file *MSC006-1.lop*. For reference, the resulting clauses are shown below (Table 4.1). One can easily identify the clauses for transitivity, symmetry and the other properties of the relations. Please note that the clauses are written in LOP-syntax[5], a PROLOG-like syntax, i.e., a clause with positive atoms $P_1, \ldots, P_n$ and negative atoms $L_1, \ldots, L_m$ is written as

$$P_1; \ldots; P_n \quad \texttt{<-} \quad L_1, \ldots, P_m.$$

with $m, n \geq 0$. Our theorem (p or q is total) has been negated and thus being transformed into two clauses. The all-quantified variables in the theorem have been

[4] In your SETHEO distribution this file as well as the other examples described in this manual can be found in the directory `$SETHEOHOME/examples`.

[5] The full definition of the LOP syntax can be found in Chapter 6.

converted into the constants c_1,...,c_4 by Skolemization. Please note that all variables which occur in the clauses (they are implicitly all-quantified) only range over one clause.

Although our formula now looks rather like a PROLOG program, it actually is a non-Horn formula, as can be seen in the fourth clause. This clause contains two positive atoms, separated by a ;. A Horn-clause on the contrary, only may contain at most one positive literal.

```
p(X,Z) <- p(X,Y),p(Y,Z).
q(X,Z) <- q(X,Y),q(Y,Z).
q(Y,X) <- q(X,Y).
p(X,Y) ; q(X,Y) <-.
<- p(c_1,c_2).
<- q(c_3,c_4).
```

Table 4.1: LOP formula for the *non-obviousness problem*

4.1.2 Running SETHEO

Given a set of clauses in the LOP-notation (Table 4.1; either generated by hand or by **plop**), we are ready to start the prover itself. In our example, the clauses are in the file *MSC006-1.lop*. Please, type **setheo MSC006-1** (no extension) on your computer to start SETHEO. Now a lot of things happen (hopefully no "command not found" or "core dumped") and a huge amount of output is printed on the screen. Here it is advisable to use a command-screen with a scroll bar.

Of most interest for us at the moment is the message

```
***** SUCCESS *****
```

which should appear after a few seconds on the screen. This means that SETHEO could refute the set of clauses, given in the file *MSC006-1.lop*, or equivalently, a proof has been found for our theorem.

An abbreviated copy of the messages on the screen (running on a very slow machine) is given in following (missing lines are denoted by ...). For reference in the text, the lines are numbered (the line numbers are not visible on your screen)[6].

```
01- inwasm V3.2.5 Copyright TU Munich (April 7, 1995)
02- command line: inwasm -cons -foldup MSC006-1
03- Parsing
04- Preprocessing:
        ...
05- Codegeneration.
06- MSC006-1.s generated in  0.07 seconds
```

[6]Depending on your version of SETHEO, several messages can be stated differently, or different numbers may be present. This, however, is quite normal.

```
07- wasm V3 Copyright TU Munich (March 25, 1994)
         ...
08- SAM V3.3 Copyright TU Munich (December 22, 1995)
09- Options : -dr -cons -dynsgreord 2 MSC006-1
         ...
10- Start proving...
         ...
11- -d:    2  time     < 0.01 sec  inferences =        9  fails =       15
         ...
12- -d:    7  time =     0.62 sec  inferences =     6410  fails =     7741

13-                  ******** SUCCESS ************

14- Number of inferences in proof      :       20
15-          - E/R/F/L                 :       17/       2/       1/       0
         ...
16- Number of unifications             :     8465
         ...
17- Instructions executed              :    32093
18- Abstract machine time (seconds)    :     0.90
19- Overall time          (seconds)    :     1.05
```

The SETHEO system processes the formula in three distinct steps:

1. The input formula is being preprocessed (see Tutorial 2 and Section 5.4.2 for details), and assembler code for the SETHEO abstract machine is generated (lines 1–6).
2. Then the code is assembled (line 7).
3. Finally, the interpreter for the abstract machine SAM is started. This module actually tries to find the proof (lines 8–19).

Each of the modules (which are separate programs) can be called with a variety of command-line parameters. If you invoke the **setheo-**command, default settings are taken. For a description of the parameters and their influence see our Tutorial 2 and Chapter 5.

After each of the modules has finished its work, it prints a line which indicates the run-time of that module (e.g., line 6 and 19). For the prover itself, we additionally give the time needed to *search* for the proof (line 18). This run-time does not take into account the overhead for book-keeping tasks, for loading the formula and for printing the statistics.

The amount of search needed to find the proof can be seen at three different points:

line 11–12 The prover does "iterative deepening". This means that the search space is only explored until a certain limit is reached. If within these limits no proof can be found, the limit is increased and the search starts over again. Here we limit the depth (`-d`). For each level the amount of time spent, the number of

inference steps (within the calculus) and the number of backtracking steps is given.

line 16 This entry gives the number of successful unifications which have been made during the search.

line 17 This number is the amount of low-level abstract machine instructions executed to find the proof. This number is the most fine-grain measure of the work done by the SAM. Please note, however, that the execution time for each instruction differs.

The actual proof found by SETHEO consists of 20 (Model Elimination) inference steps (line 14). Their type and frequency are given in line 15. In order to unstand these figures and the proof itself, some knowledge about the Model Elimination Calculus is necessary. In the next section, we will give a short (informal) introduction into the calculus. After that, we will have a detailed look at the proof generated by SETHEO.

4.1.3 The Model Elimination Calculus

Historically, Model Elimination was introduced as a two-sorted variant of resolution (see, e.g., [Loveland, 1978]). Proof-theoretically, however, Model Elimination can also be viewed as a specialized tableau method, the so-called *connection tableau calculus* [Letz *et al.*, 1992; Letz *et al.*, 1994].[7] A connection tableau for a set of clauses is generated by first applying the *start rule* and then repeatedly applying either the *reduction* or the *extension rule*. Employment of the start rule means selecting a clause from a set of possible start clauses and attaching its literals to the root node; the start rule is the standard tableau expansion rule. The reduction rule permits the closing of a branch (or *path*) by unifying a subgoal[8] K with the complement of a path literal L — L is called a *connected path literal*; the reduction rule is the standard closure rule of free-variable tableaux. The extension rule requires the performing of a unification step by attaching the (instantiated) literals of a *connected clause* to the current subgoal K, i.e., a clause that contains a literal which can be unified with the complement of K. The extension rule, which is obviously a specific combination of tableau expansion and reduction, implements the connectedness condition.[9]

The connectedness condition leads to the *goal-directedness* of the calculus, since every literal in the tableau bears a relation to the start clause, which needs not be the case for general tableaux. On the other hand, however, the connectedness condition causes the *non-confluence* of the calculus, i.e., if an open tableau branch cannot be expanded, it does not guarantee the satisfiability of the input clauses.[10] As a consequence, when searching for a proof, one has to enumerate *all* connection tableaux. The most efficient way of tableau enumeration is by using iterative deepening search procedures with

[7] One major argument for this interpretation is that, when viewed as a tableau method, Model Elimination (in its weak variant) is *cut-free*.

[8] A *subgoal* is the literal at the leaf of an open branch.

[9] SLD-resolution can be viewed simply as the connection tableau calculus without reduction steps. This shows that SLD-resolution is inherently cut-free and not a cut calculus like general resolution.

[10] For this reason, in contrast to general tableaux, connection tableaux (and hence Model Elimination) are not suited for model *generation*.

backtracking (see [Stickel, 1988]).

4.1.4 Looking at the Proof

The graphical Display

The LaTeX Output

proof for MSC006-1

Table 4.2: proof for MSC006-1

4.2 Tutorial 2

For this tutorial, we assume the reader to be familiar with the basics of the Model Elimination Calculus and SETHEO. This tutorial covers the influence of refinements and extensions of the basic Model Elimiantion Calculus on the search behavior of SETHEO. Again, we use an example to illustrate the techniques implemented in SETHEO, in fact we use the same example as in the previous tutorial. For reference, its clauses (in LOP syntax) are given below in Figure 4.3. This example is, of course, no challenge problem for Automated Theorem Provers (it used to be one about 10 years ago), but it has some nice properties.

```
(1)  p(X,Z) <- p(X,Y),p(Y,Z).
(2)  q(X,Z) <- q(X,Y),q(Y,Z).
(3)  q(Y,X) <- q(X,Y).
(4)  p(X,Y) ; q(X,Y)<-.
(5)  <- p(a_1,a_2).
(6)  <- q(a_3,a_4).
```

Table 4.3: LOP clauses of the Example (file: MSC006-1.lop)

For our first experiment, we run SETHEO, using the basic Model Elimination Calculus, and performing depth-first iterative deepening over the depth of the tabelau (A-literal depth)[11]. Such a run will be accomplished using the two commands:

```
inwasm MSC006-1
sam -dr MSC006-1

SUCCESS
SETHEO-output for MSC006-1.lop
```

Figure 4.1: Output of the SAM for MSC006-1.lop

After some time, SETHEO finds a proof, consisting of 18 inferences, two of which are Model Elimination Reduction steps (see Figure 4.1 for SETHEO's output). A graphical representation of the tree can be obtained by using the tool **xptree** (see

[11]Using different bounds will be discussed below in Section 4.7.

Chapter 5 for details on **xptree**). For us of interest now, however, is not the proof itself, but the amount of search involved to find the proof. As shortly mentioned already in Tutorial 1, this is reflected in the following sizes and numbers:

Abstract Machine Time This number gives the amount of time (CPU user time), the SAM, SETHEO's Abstract Machine, needed to find the proof. However, on different machines, and even on different runs on the same machine, this value may vary due to load-fluctuations of the machine and due to inaccuracies of the time measuring. Therefore, this figure should be taken as a first approximation only.

Total number of inferences n_i gives the overall number of times, a unification was tried during search (by trying to perform an Extension or Reduction step). This number directly reflects the amount of search performed, but does not take into account, how long each attempted unification takes.

Number of Fails n_f is the number of times, a unification failed, or a bound has been reached. This number is again splitted into those two values.

The output SETHEO produces at the end of the run gives these values for the overall search. Additionally, these values are also printed, after the entire search space has been exhausted with a given bound. Figure 4.1 shows these figures for the increasing depth-bound (`-d`). As can be seen clearly, the size of the search space grows substantially with each iteration (at least exponentially).

In the following, we present several improvements of the Model Elimination Calculus and SETHEO's search procedure and have a look at these figures. The list of improvements discussed in this tutorial is only a small selections of techniques integrated into SETHEO. For a list of all techniques and the corresponding command-line switches see Chapter 5. Furthermore, we do *not* present the theoretical background nor any formal definitions or theorems. For such issues see, e.g., [Letz *et al.*, 1992; Letz *et al.*, 1994; Letz, 1993; ?].

4.3 Static Refinements

4.3.1 Regularity

One of the most powerful refinement of the pure Model Elimination Calculus is its restriction to *regular* tableaux.

Definition 4.3.1 (regular tableau) *A Model Elimination Tableau $\mathcal{T}$ is regular, if and only if on each path from the root to a leaf, no literal occurs more than once.*

One can show (cf. [Letz *et al.*, 1992; Letz *et al.*, 1994]) that for each closed tableau there also exists a closed regular tableau, i.e., we don't loose any proofs if we are searching for regular tableaux only.

Figure 4.2 contains two tableaux for the same subgoal (taken from our example). The left tableau is not regular, because the literal $\neg p(a, b)$ occurs twice in it; the right tableau is regular. In our case, it is easy to see, why the left tableau is not regular: in

attempt to solve the goal $\neg p(a, b)$, an extension step into the symmetry clause (clause number (3)) is made, yielding a new subgoal $\neg p(b, a)$. Then this clause is used again to obtain $\neg p(a, b)$. Comparing this subgoal to the original one, we have gained nothing! Therefore, we can leave these two steps out, yielding a regular tableau. Restricting the search to certain kinds of tableaux thus reduces the search space considerably.

-figure-

Figure 4.2: Model Elimination Tableau and Regular Model Elimination Tableau

Within SETHEO, there are two different ways to enforce regularity: a *direct regularity check* which is performed, as soon as an Extension or Reduction step is tried, and the generation of *regularity constraints*.

Direct Regularity Check. This check can be activated by calling **inwasm** with the option `-eqpred`, yielding the following sequence of commands:

```
inwasm -eqpred MSC006-1
sam -dr MSC006-1
```

The direct regularity check is realized as an *equal-predecessor* check, i.e., when an Extension or Reduction step is tried the path is searched for equal literals. Note that this is only a restricted regularity check since *later* instantiations to equal literals are not detected.

When we look at the result SETHEO produces, a considerable reduction in the amount of search necessary to find the proof can be seen. Table 4.4, lines 1 and 2 shows typical figures, compared to the basic calculus.

Method	t_{SAM}	n_i	n_f	n_i^3	n_i^4	n_i^5	n_i^6	n_i^7
basic	2.34	160606	25252	25	145	2525	26262	282828
`-eqpred`	12.40	117645	113085	30	110	814	7492	109190
`-reg`	4.37	35557	45162	30	108	703	5037	29670

Table 4.4: Search space for Basic Model Elimination (line 1), and implementations of regularity. n_i is the total number of inferences, n_f the total number of fails, and n_i^k the number of inferences on search level k.

Regularity Constraints. A more elegant and powerful method, which provides a full regularity check, is realized by syntactical inequality constraints. These constraints of the form $[X \notin \{t_1, \ldots, t_n\}]$ are attached to variables (here X) and checked permanently. As soon, as X gets bound to a term, its constraints are evaluated. If a violation occurs (i.e., X gets instantiated to one of the t_i), backtracking occurs within the SAM Abstract Machine. Looking at our above example of Figure 4.2 the regularity-checker would generate the constraint $[X, Y] \neq [b, a]$. This constraint would fire immediatedly, if the symmetry clause was tried for the second time, causing the SAM to fail and backtrack.

The results for running SETHEO with regularity constraints, which is performed by the commands

```
inwasm -reg MSC006-1
sam -dr -reg MSC006-1
```

are shown in Table 4.4. Although the figures for this example are quite similar for these two methods of enforcing regularity, it is advisable to use regularity constraints, because it is (a) more powerful, but costs a little more overhead, and (b) neatly fits into other Calculus Refinements described below. As a general hint, it is *always* advisable to activate the enforcement of regular tableaux. Note, that the default parameters for **setheo** incorporate this option.

4.3.2 Tautology and Subsumption Constraints

A Model Elimination Tableau can be further restricted, namely that no instance of a clause in the tableau is a *tautology*, and that no instance of a clause is subsumed by another clause. Again, when these restrictions are enforced, no proofs are lost. In SETHEO, these conditions are checked permanently using the constraint mechanism. Constraints for checking for tautology and sumbsumption are generated during the run of the **inwasm** compiler, if it is invoked with the **-cons** or **-taut -subs** options. Looking at our example, we can easily detect instantiations of clauses which will result in tautological clauses or clauses which are subsumed by others. Let's have a look at clause (4) in Figure 4.3 which expresses the symmetry of p. If we instantiate X to the same value as Y (e.g., by calling this clause with a subgoal $\neg p(a,a)$), we obtain the following instatiation of that clause in the tableau `p(X,X) <- p(X,X)`. Obviously, this clause is tautologial and its application does not lead us anywhere. Therefore, we can *forbid* such an instantiation by adding the constraint `[X] =/= [Y]` to the clause.

In the case of the transitivity clauses (e.g., clause (1)), we obtain a similar sitation: If two of the variables in that clause X, Y, Z get instantiated to the same value, the resulting clause is tautological, namely if $X = Y$ or $Y = Z$. This can be forbidden, using two constraints.

Additionally, there exist instantiations, where this clause is subsumed by clause (5) `<- p(a_1,a_2)`, namely X is instantiated to c_1 and Y to c_2, or Y to c_1 and Z to c_2. In both cases, the resulting instance of the clause is subsumed by clause (5). Therefore, the following constraints are automatically added to clause (1):

```
p(X,Z) <-  p(X,Y), p(Y,Z)
    : [Y] =/= [Z]          /* tau      */,
      [Y] =/= [X]          /* tau      */,
      [X,Y] =/= [a_1,a_2]  /* sub by 5 */,
      [Y,Z] =/= [a_1,a_2]  /* sub by 5 */.
```

A list of all generated constraints can be obtained, when the **inwasm** is called with the option `-lop`. In that case, a file *file*`_pp.lop` is generated which contains all contrapositives of all clauses, the generated constraints as well as other information.

A variety of other refinements, working with constraints have been developed and integrated into SETHEO (e.g., overlapping), but these will not be discussed here. For

details see Chapter 5 and the Glossary. Comparative run-times with our example is shown in Table 4.5[12].

Method	t_{SAM}	n_i	n_f	n_i^3	n_i^4	n_i^5	n_i^6	n_i^7
basic	2.34	160606	25252	25	145	2525	26262	282828
`-taut`	2.03	16411	21166	30	105	396	1946	13925
`-subs`	2.34	160606	25252	25	145	2525	26262	282828
`-subs -taut -reg`	2.32	16325	21052	30	101	380	1880	13925

Table 4.5: Search space for Basic ME and several comile-time refinements. `-cons` is the abbreviation for `-reg -taut -subs -anl`.

4.3.3 Deletion of Links

The search space for finding a Model Elimination proof is spanned by the possible *connections* between literals of the clauses. The more connections, the larger the search-space (which increases at least exponential with their number). Therefore, any method which can remove connections without sacrificing completeness of the search procedure can lead to a considerable decrease in run-time needed to find a proof.

The method built into SETHEO tries to remove connections which are of no use. This option can be activated by calling **inwasm** with `-linksubs` or `-rlinksubs`[13]. In this tutorial, wo won't describe in detail, how this preprocessing step works. Rather, we explain what happens with selected clauses of our example. Let us consider clause (3) (Symmetry of q). The second literal (the tail literal) has several connections: some going to positive q literals of other clauses, and one to the head-literal of our clause (3). The latter connection, we call it a "back-connection", is of interest. If clause (3) has been called from a subgoal $\neg p(a, b)$, then our tail literal gets $\neg p(b, a)$. If we now follow the back-connection, we end up with a new subgoal $\neg p(a, b)$ which is exactly our original one. In this case, following the back-connection does not make any sense[14]. Thus this connection can be removed without harming completeness.

Within the rules of transitivity (clauses (1) and (2)), one back-connection in each clause can be removed as well. Transitivity clauses are in particular harmful, because they have two subgoals (the longer a clause, the more search space it induces), and these subgoals always introduce new variables (in our case variable Y).

Results of experiments with deletion of links are shown in Table 4.6[15].

[12]For activating these constraints within the SAM, the command **sam** must be called with the option `-st`

[13]Note that this option has to be set for the **inwasm** only.

[14]In this case, our situation can be detected by the regularity-contraints as well, but in general, these two methods are independent from each other.

[15]The option `-linksubs` differs from `-rlinksubs` in such a way that `-rlinksubs` ignores connections into single-literal clauses (facts), thus reducing the required run-time to find removable clauses.

Method	t_{SAM}	n_i	n_f	n_i^3	n_i^4	n_i^5	n_i^6	n_i^7
basic	2.34	160606	25252	25	145	2525	26262	282828
`-linksubs`	52.42	564876	722977	27	81	556	6437	557766
`-linksubs -cons`	0.87	4634	4809	27	72	213	868	3445

Table 4.6: Search space for Basic ME, removal of connections, and removal of connections combined with constraints

4.4 Dynamic Constraints: Antilemmata

The search mechanism of basic Model Elimination is based on enumerating all possible solutions for the involved subgoals. Since in general variables are shared, solutions found for one subgoal influence the solvability of the still unsolved subgoals. Thus, the solution of an early subgoal may cause a fail in a later subgoal. In that case, a new solution must be computed for the first subgoal.

If, however, this solution is the same as the previous one, nothing could be accomplished, because again it won't be possible to solve the later subgoal. Therefore, this situation must be detected and must lead to a fail. If the recomputation of a solution appears very often, a considerable amount of redundancy in the search is visible. See Figure 4.3 for a simple example.

Figure 4.3: A simple example for the necessity of Antilemmata: Basic ME will compute the solution $X = a$, detect a fail and backtrack for 100 times, before the solution $X = b$ can be computed.

The refinements of the Model Elimination Calculus described in the previous sections cannot prevent the prover from computing the same solution over and over again. Thus, the concept of *Antilemmata* had been developed and implemented to avoid the repetition of solutions: During backtracking all instantiations of variables which are undone are converted into Antilemmata. These forbid to compute the same instantiation again. The repetition-check is performed every time when a further Extension or Reduction step is tried (see Figure 4.4).

Antilemmata can be invoked with the commands

```
inwasm MSC006-1
sam -dr -anl MSC006-1
```

The performance gain by Antilemmata is shown in Table 4.7.

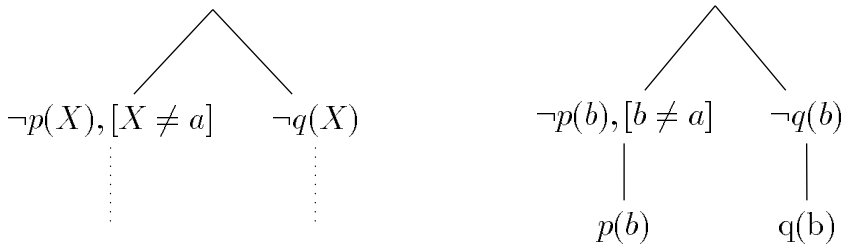

Figure 4.4: The use of Antilemmata leads to the generation of an Antilemma-Constraint $X \neq a$ after the first fail. So after backtracking recomputation of the solution $X = a$ is avoided. Instead immediately the the solution $X = b$ is obtained.

Method	t_{SAM}	n_i	n_f	n_i^3	n_i^4	n_i^5	n_i^6	n_i^7
basic	2.34	160606	25252	25	145	2525	26262	282828
`-anl`	15.12	117727	195984	30	127	1625	15812	100124
`-cons`	1.17	5350	6389	30	100	359	1321	3531
`-cons -linksubs`	1.17	3879	3996	27	72	213	804	2754

Table 4.7: Search space for Basic ME, and for ME with Antilemmata

4.5 Additional Inference Rules: Folding Up and Folding Down

A major source of redundancy in the Model Elimination Calculus is the necessity to solve the same subgoals over and over again. Not only during the search, the same proofs for subgoals occur repeatedly, but also in the final tableau, the same subgoal can show up more than once. When performing a proof manually, usually, the human defines a *lemma*, proofs it and then use it for each occurrence of the subgoal.

A similar approach can be made for the Model Elimination Calculus as well ("Lemmaizing", cf. [?]). As soon as a subgoal gets solved, the solution is stored in a so-called lemma store from where it can be used when the same or similar subgoals have to be solved later on.

This approach[16], however, carries several severe problems: during the search a huge amount of lemmas are generated and must be stored in such a way that they can be retrieved easily. An unrestricted use of all these lemmas usually lead to an explosion of the search space. Therefore, effective methods for selecting "good" lemmas would be necessary.

In order to avoid these problems, we have implemented a restricted version of lemmas by the means of additional inference rules. In contrast to "real lemmas" our restriction means that

- no lemma survives backtracking over subgoal which caused its generation,
- variables within a lemma can be bound once only.

[16]The basic functionality for generating and using lemmata has been built into SETHEO by means of built-in predicates. These are described in Chapter 7, but they have to be added manually to the input formula.

Nevertheless our restricted version is correct and sound and requires only a minimal overhead within the SAM (actually, only a few pointers need to be adjusted).

Figure ?? shows with the help of a small example, how this mechanism for "folding up" works: let us assume, subgoal $\neg p(a,b)$ just has been solved (The triangle below that subgoal indicates a closed tableau), $p(a,b)$ now can be used as a lemma which could be used, if similar subgoals (in our example $\neg p(X,b)$) have to be solved later on. Instead of copying $p(a,b)$ into an extra piece of memory, we just add $p(a,b)$ as a new node to the branch from the root of the tableau to the current nodes (right hand side of the Figure). This can be accomplished by adjusting a few pointers. Then, when a new subgoal (in our case $\neg p(X,b)$) has to be solved, the branch can be simply closed by one ME reduction step (dotted line).

`In the Non-Horn case,`

`figure`

`position of folded up literal`

`folding down`

Method	t_{SAM}	n_i	n_f	n_i^3	n_i^4	n_i^5	n_i^6	n_i^7
basic	2.34	160606	25252	25	145	2525	26262	282828
`-cons`	1.17	5350	6389	30	100	359	1321	3531
`-foldup -cons`	1.33	7474	8797	30	100	365	1465	5505
`-foldupx -cons`	1.10	5324	6197	30	100	365	1318	3502
`-folddown -cons`	1.60	8402	9837	38	126	446	1739	6042

Table 4.8: Search space for Basic ME, ME with refinements, and refinements combined with the folding-down and folding-up inference rule. The proof with folding up contains 1 folding step, whereas in the experiment with folding down,the additional inference rule could not be used at all.

4.6 Reordering

The search space of model elimination can be seen as a complex form of an and-or-tree in which tableaux are and-nodes (connecting the contained subgoals) and subgoals are or-nodes (connecting the possible unification partners). While, for any subgoal, all possible unification partners have to be tried in order to guarantee completeness, soundness requires to solve all subgoals. The order of clause and subgoal selection strongly influences the size of the search space.

4.6.1 Clause Reordering

In general, for a subgoal several unification partners are available. These are connected literals, in the non-Horn case stemming from the path or a connected input clause and in the Horn case stemming from a connected input clause. Together they built a so-called *choice point*. The unification partners within a choice point are *alternatives*, i.e., only one of them has to appear in the closed tableau. Therefore, alternatives that

will probably not lead to a fail should be preferred. This results in the *early-success* principle for clause selection[17].

Since the number of open subgoals usually indicates the size of the search space, short clauses should be preferred rather than long ones.[18] This results in the *shortest-clause* principle.

Connected clauses are variants of input clauses, not sharing variables with the tableau. Therefore, a dynamic clause selection yields the same order of alternatives than a static clause selection. However, connectedness itself cannot be determined statically since, e.g., statically, a unit clause $p(a)$ is connected with a subgoal $\neg p(X)$ whereas, dynamically, $p(a)$ might not be connected with the instance of $\neg p(X)$. Therefore, statically only *weak connections* can be computed, i.e., a superset of the connected clauses.

The ordering of the input clauses is performed as follows in SETHEO. Statically, for each subgoal K and each connected literal in the input clauses, a so-called *contrapositive* of this input clause is inserted into the choice point of K, i.e., a variant of the clause with the connected literal as *head* and the other literals as subgoals. The contrapositives that are connected to a subgoal are ordered by their lengths, due to the shortest-alternative principle. If several contrapositives have the same length, the ones with lower complexity and more free variables are preferred. In the proof search, connected path literals are preferred to contrapositves, i.e., reduction steps are tried before extension steps.

Static clause reordering is performed by default. It can be suppressed by calling

```
inwasm -noclreord MSC006-1
```

4.6.2 Subgoal Reordering

A clause is proven if all its have been solved. In order to retract an unprovable clause as early as possible, the solutions of a subgoal should be exhausted as early as possible. Therefore, subgoals for which probably only a few solutions exist should be selected earlier than subgoals for which many solutions exist. This results in the *fewest-solutions* principle for subgoal selection. The special case of the fewest-solutions principle without solutions is called *first-fail* principle.

Subgoal selection can either be invoked *statically* to determine the order of subgoals or *dynamically*. The static version is cheaper (in terms of CPU-time) than the dynamic because all variants of the input clauses are handled simultaneously. However, static subgoal selection cannot unambiguously determine the order of subgoals as can be seen in the following example.

Consider the transitivity clause $p(X,Z) \leftarrow p(X,Y), p(Y,Z)$. Statically, none of the subgoals can be preferred wrt. the above principles. Dynamically, performing an extension step with the transitivity clause at a literal $\neg p(a,Z)$ leads to the preference of

[17] The term 'clause selection' has been established for the selection of alternatives from a choice point though choice points usually contain as well clauses as path literals.

[18] The *length of a clause* is its number of subgoals.

the first subgoal $\neg p(a, Y)$ due to the fewest-solutions principle, performing an extension step at a literal $\neg p(X, b)$ leads to the preference of the second subgoal $\neg p(Y, b)$.

SETHEO performs a static and a dynamic ordering of the subgoals in a contrapositive. The static selection criteria are described in [Letz *et al.*, 1992]. During the static subgoal reordering declarative and procedurale clauses are distinguished. By default, the subgoals of declarative clauses are reordered and the subgoals of procedurale clauses are not reordered. To enforce the static subgoal reordering of procedurale clauses, call

```
inwasm -sgreord MSC006-1
```

To suppress the static subgoal reordering of procedurale clauses, call

```
inwasm -nosgreord MSC006-1
```

Table 4.9 shows the results for basic ME, ME with static clause reordering, ME with static subgoal reordering and ME with static clause and subgoal reordering.

Method	t_{SAM}	n_i	n_f	n_i^3	n_i^4	n_i^5	n_i^6	n_i^7
basic	2.34	160606	25252	25	145	2525	26262	282828
`-nosgreord -noclreord`	$0.70^{\dagger}$	5848	8128	30	99	409	3467	1804
-"- `-cons`	$0.80^{\ddagger}$	3225	4335	30	94	379	2070	643
`-nosgreord -cons`	2.17	12857	16513	30	94	379	2070	10275
`-noclreord -cons`	$0.73^{\dagger\dagger}$	2961	3627	30	100	359	1321	1142

Table 4.9: Search space for Basic ME, ME with static clause reordering, ME with static subgoal reordering and ME with static clause and subgoal reordering. Proof † is different from the others and contains 46 inferences (‡ with 34 inferences, †† with 27 inferences) in contrast to the "standard proof" with 21 inferences.

Currently, in SETHEO dynamic subgoal selection is performed in a depth-first manner only, i.e., between the literals of the contrapositive attached in the last extension step.[19] To realize dynamic subgoal selection two criteria have been implemented which approximate the fewest-solutions principle.

1. Select the subgoal with the highest complexity.
2. Select the subgoal with the least free variables.

Optionally, we can use an additional proviso, namely, not to switch to a subgoal with a different predicate symbol, according to the order of subgoals computed statically. These criteria result in six functions for dynamic subgoal selection (see Figure 4.10).

These selection strategies can be invoked by

```
sam -dynsgreord <number> MSC006-1
```

where number is out of 1,...,6 and indicates one of the selection functions f_1,...,f_6.

[19] Therefore, currently SETHEO is really a model elimination prover.

	with proviso	without proviso
Select due to criterion 1.	f_1	f_4
If one of the subgoals is ground, ignore all non-ground subgoals. Select due to criterion 1. If more than one subgoal is selected, select from these due to criterion 2.	f_2	f_5
Select due to criterion 2. If more than one subgoal is selected, select from these due to criterion 1.	f_3	f_6

Table 4.10: SETHEO's selection functions for traditional dynamic subgoal selection

Table 4.11 shows the results for basic ME and for ME with dynamic subgoal reordering.

Method	t_{SAM}	n_i	n_f	n_i^3	n_i^4	n_i^5	n_i^6	n_i^7
basic	2.34	160606	25252	25	145	2525	26262	282828
`-cons`	1.17	5350	6389	30	100	359	1321	3531
`-dynsgreord 1 -cons`	1.38	6479	8050	30	100	362	1450	4528
`-dynsgreord 2 -cons`	1.38	6479	8050	30	100	362	1450	4528
`-dynsgreord 3 -cons`	1.38	6479	8050	30	100	362	1450	4528
`-dynsgreord 4 -cons`	1.38	6479	8050	30	100	362	1450	4528
`-dynsgreord 5 -cons`	1.38	6479	8050	30	100	362	1450	4528
`-dynsgreord 6 -cons`	1.38	6479	8050	30	100	362	1450	4528

Table 4.11: Search space for Basic ME and ME with dynamic subgoal reordering

4.6.3 Subgoal Alternation

One common principle of standard backtracking search procedures in model elimination (and in Prolog) is that, whenever a subgoal has been selected, its choice point must be completely finished, i.e., when retracting an alternative in the choice point of a subgoal, one has to stick to the subgoal and try another alternative in its choice point. This standard methodology has a deep search-theoretic weakness that has not been recognized so far.

The weakness of remaining in the same choice point can be illustrated with the following generic example, variants of which often occur in practice. Given the subgoals $\neg p(X,Y)$ and $\neg q(X,Y)$ in a tableau, assume the following clauses be in the input.

$$
\begin{array}{l}
p(a,a), \\
p(X,Y) \vee \neg p'(X,Z) \vee \neg p'(Y,Z), \\
p'(a_i,a), \quad 1 \leq i \leq n, \\
q(a_i,b), \quad 1 \leq i \leq n.
\end{array}
$$

Suppose further we have decided to select the first subgoal and perform depth-first subgoal selection. The critical point, say at time t, is after the unit clause $p(a,a)$ in the choice point was tried and no compatible solution instance for the other subgoal

was found. Now we are forced to enter the clause $c = p(X,Y) \vee \neg p'(X,Z) \vee \neg p'(Y,Z)$. Obviously, there are n^2 solution substitutions (unifications) for solving the clause c (the product of the solutions of its subgoals). For each of those solutions we have to perform n unifications with the q-subgoal, which all fail. This amounts to a total of $1 + n^3$ unifications. Observe now what would happen when at time t we would not have entered c, but would switch to the q-subgoal instead. Then, for each of the n solution substitutions $q(a_i, b)$, one would jump to the p-subgoal, enter c and perform just n failing unifications for its first subgoal. This sums up to a total of just $n(1 + n)$ unifications.

It is apparent that this phenomenon has to do with the fewest-solutions principle. The clause c generates more solutions for the subgoal $\neg p(X,Y)$ than the clauses in the choice point of the subgoal $\neg q(X,Y)$. This shows that taking the remaining alternatives of *all* subgoals into account provides a choice which can better satisfy the fewest-solution principle. The general principle of *subgoal alternation* is that one always switches to that subgoal with a next clause that produces the fewest solutions.

The question is, when it is worthwhile to stop the processing of a choice point and switch to another subgoal? As a matter of fact, it cannot be determined in advance, how many solutions a clause in the choice point of a subgoal produces for that subgoal. A useful criterion, however, is the *shortest-clause* principle, since, in the worst case, the number of subgoal solutions coming from a clause is the product of the numbers of solutions of its subgoals.[20]

In summary, subgoal alternation works as follows. The standard subgoal selection and clause selection phases are melted together into one single selection phase that is performed before each derivation step. The selection yields the subgoal for which the most suitable unification partner exists wrt. the number of solutions probably produced. For this, the unification partners of all subgoals are compared with each other using, for instance, the shortest-clause principle. If more than one unification partner is given the mark of 'best', their corresponding subgoals have to be compared due to the principles for standard subgoal selection, namely the first-fail principle and the fewest-solutions principle.

In order to compare the working of subgoal alternation (using the shortest-clause principle) with the standard non-alternating variant, consider two subgoals A and B with clauses of lengths 1,3,5 and 2,4,6 in their choice points, respectively. Table 4.6.3 illustrates the order in which clauses are tried.

To use SETHEO with subgoal aternation, call, e.g.,

```
sam -singlealt MSC006-1
```

or

```
sam -multialt MSC006-1
```

There are different refinements of subgoal alternation each of which is suitable for special problem characteristics. See Section 5.4.4 how to invoke the different refine-

[20] Also, the number of variables in the *calling* subgoal and in the *head* literal of a clause matter for the number of solutions produced.

standard backtracking	subgoal alternation
A1 B2	A1 B2
A1 B4	A1 B4
A1 B6	A1 B6
A3 B2	⇄ B2 A3
A3 B4	B2 A5
A3 B6	⇄ A3 B4
A5 B2	A3 B6
A5 B4	⇄ B4 A5
A5 B6	⇄ A5 B6

Figure 4.5: Order of tried clauses for subgoals A and B with clauses of lengths 1,3,5 and 2,4,6 in their choice points, respectively. ⇄ indicates subgoal alternations.

ments of subgoal alternation. Table 4.12 shows the results for basic ME and for ME with subgoal alternation.

Method	t_{SAM}	n_i	n_f	n_i^3	n_i^4	n_i^5	n_i^6	n_i^7
basic	2.34	160606	25252	25	145	2525	26262	282828
`-singlealt`	2.34	160606	25252	25	145	2525	26262	282828
`-multialt -cons`	2.73	13158	11364	35	138	634	3372	8970

Table 4.12: Search space for Basic ME and ME with subgoal alternation

4.7 Bounds

Until now, we only have considered the depth-bound (A-literal depth) and iterative deepening over that bound as a means to obtain completeness. SETHEO, however, features a large variety of bounds. In general, the bounds in SETHEO follow the rules:

- Each bound can be used separatedly or combined with other bounds.
- Iterative deepening is performed with one bound at each time only.
- Several bounds do not terminate when used alone (e.g., term complexity, or maximal number of open subgoals).
- Bounds can be obtained and set using specific LOP built-in predicates. These predicates are described in Chapter 7.

For those bounds which allow for iterative deepening, a fixed start-value is used. The increment can be set using the command-line options. Table 4.13 shows a short defintion of all bounds available in the current version and their main features.

Most commonly used are the depth bound, the inference bound, and the weighted depth bound which has been designed to combine the advantages of the depth and inference bound. The choice of a bound dramatically influences the beaviour of

Name	option	terminate?	iterate with	type
depth	`-d`	Y	`-dr`	local
inference	`-i`	Y	`-ir`	global
clause dependent depth	`-cdd`	Y	`-cddr`	local
weighted depth	`-wd`	Y	`-wdr`	weakly local
term complexity	`-tc #`	N	-	-
depth dependent term complexity	`-tcd[12] #`	N	-	-
free variables	`-fvars #`	N	-	-
open subgoals	`-sgs #`	N	-	-

Table 4.13: Search bounds for SETHEO

SETHEO, as is depicted in Table 4.14. However, there does not seem to be a "universally good" iteration bound for all possible problems.

	t_{SAM}	n_i	n_f	$b_i : n_i$	$b_i : n_i$	$b_i : n_i$	$b_i : n_i$	$b_i : n_i$	$b_i : n_i$
`-dr`	1.17	5350	6389	3:30	4:100	5:359	6:1321	7:3531	–
`-ir`	68.08†	296043	365191	5:69	8:629	11:5369	14:29458	17:139963	20:120555
`-cddr`	1.20	5560	6570	5:71	6:182	7:342	8:789	9:1733	10:2400
`-wdr`	4.52	18093	22926	3:30	4:94	5:316	6:949	7:3735	8:12960

Table 4.14: Search space for different completeness bounds. $b_i : n_i$ gives the total number of inferences (unifications) performed with bound b_i. † is the shortest proof with 20 inferences

Therefore, an appropriate completeness bound must be selected upon information about similar proof tasks within the same domain, and possible additional information about the proof tasks.

Another approach to deal with this problem is to explore several different completeness bounds in parallel in a competitive way (as e.g., as SiCoTHEO does): on each processor, SETHEO tries to solve the problem, using a different bound (or a combination of bounds). The processor which finds a solution first, wins and stops the other processors. For details on this approach see [?].

4.7.1 Locality and Globality of Completeness Bounds

Typically, deduction enumeration procedures are implemented by *decrementing* the currently available resource when performing inference steps, and backtracking when the resource is exhausted. For certain completeness bounds, the *available* resources can be directly assigned to the subgoals in the tableau, at the time when an inference step is to be performed at a subgoal. Some of these completeness bounds have the *locality* property.

Definition 4.7.1 (Local and global completeness bounds) *Given a subgoal S, let n be the resource available for the predecessor of S at the time the subgoal S was attached.*

- *A completeness bound $\mathcal{B}$ is called* local *if there is a strictly monotonicly increasing mapping r such that the resource available for solving S equals $r(n)$.*

- *A completeness bound $\mathcal{B}$ is called* weakly local *if there is a strictly monotonicly increasing mapping r such that the resource for solving S is $\geq r(n)$.*
- *If a bound is not weakly local, it is called* global.

Global bounds have disadvantages when search reduction techniques like local failure caching are applied (see [Letz and Ibens, 1996]).

4.7.2 Completeness Bounds

Here, an introduction into the aforementioned completeness bounds is given.

Inference Bound The most natural completeness bound is the *inference bound*, which identifies the bounded resource with the maximal number of inferences needed to derive a tableau. An optimal realization of this bound uses *look-ahead* information as follows. As soon as a clause is attached, the number of its subgoals is added to the current inference number, since obviously for every subgoal of the clause at least one inference step is necessary to solve it. This permits that an exceeding of the current resource limit is detected as early as possible. Obvioulsy, the inference bound is a global bound.

The inference bound was the first search bound used for depth-first iterative deepening search in model elimination [Stickel, 1988]. Experiments have demonstrated, however, that the inference bound is not the most successful strategy (consult [Letz *et al.*, 1992]). The main weaknesses are the following. Firstly, the bound is too optimistic, since it implicitly assumes that subgoals which are not yet processed may be solved with just one inference step (but cf. [Harrison, 1996] for a slight improvement). Secondly, the inference bound is *global* in the sense that brother subgoals share their inference resources, which has disadvantages concerning local failure caching (see [Letz and Ibens, 1996]).

Depth Bound A further simple completeness bound is the *depth bound*, which limits the length of the tableau branches. In connection tableaux one can relax this bound so that it is only checked when non-unit clauses are attached. This implements a certain unit preference strategy. An experimental comparison of the inference bound and the relaxed depth bound is contained in [Letz *et al.*, 1992] (compare also [Harrison, 1996]). The weakness of the depth bound is that it is too coarse in the sense that the search space to explore in each iteretive deepening level increases doubly exponentially (cf. [Letz and Ibens, 1996]). The depth bound is a local bound.

Clause Dependent Depth Bounds Using the depth bound, when a clause is attached to a subgoal with resource n, the resource of each of the new subgoals is $n-1$. A straightforward generalization of the depth bound can be obtained by allocating resources $r(n,l)$ to the new subgoals where l is the number of new subgoals. If r is strictly monotonicly increasing and $r(n,l) < n$, then the corresponding bound is obviously a local completeness bound, like the depth bound. We call such bounds *clause dependent depth bounds*. With clause dependent depth bounds a smoother increase of the iterative deepening levels can be obtained. One such bound is defined

by $r(n,l) = n - l$ (this bound is available in SETHEO since version V.3 [Goller *et al.*, 1994]). Another one, which is defined by $r(n,l) = (n-1)/l$, was used by Harrison and called *sym* [Harrison, 1996]).

Weighted-Depth Bounds Although with clause dependent depth bounds a higher flexibility can be obtained, those bounds are all in the spirit of the pure depth bound, in the sense that they are local and favour symmetric proofs. In order to increase the flexibility and permit an integration of features of the inference bound, we have developed the so-called *weighted-depth bounds*. The main idea of the weighted-depth bounds is to use a local bound (like a clause dependent depth bound) as a basis, but take the inferences into account when allocating the resource to a subgoal. In detail, this is controlled by three functions w_1, w_2, w_3 as follows. When entering a clause, the available resource n for the new subgoals is first decremented depending on the number l of subgoals of the clause, i.e., $n' = w_1(n,l)$. Then, n' is divided into two parts, a *free* part $n_\mathrm{f} = w_2(n',l)$ and an *additive* part $n_\mathrm{a} = n' - n_\mathrm{f}$. Whenever a subgoal is selected, the additive part is modified depending on the inferences Δi performed since the clause has been entered, i.e., $n'_\mathrm{a} = w_3(n_\mathrm{a}, \Delta i)$. The eventually allocated resource for a subgoal then is $n_\mathrm{f} + n'_\mathrm{a}$.

Depending on the choice of the functions w_1, w_2, w_3 the corresponding weighted-depth bound can simulate the inference bound,[21] namely by setting $w_1(n,l) = n - l$, $w_2(n',l) = 0$, $w_3(n_\mathrm{a}, \Delta i) = n_\mathrm{a} - \Delta i$, or the (clause dependent) depth bound(s) or any combination of them. A parameter selection which represents an interesting new completeness bound combining inference and depth bound is, for example, $w_1(n,l) = n - 1$, $w_2(n',l) = n' - l$, $w_3(n_\mathrm{a}, \Delta i) = n_\mathrm{a}/(1 + \Delta i)$. This bound turned out to be much more successful in practice than each of the pure bounds (see [Letz and Ibens, 1996]). One reason for the success of this strategy might be that it performs a *unit preference* strategy, which is one of the most successful general paradigms in automated deduction. This completeness bound is only weakly local.

4.8 Exercises

[21] We assume that the look-ahead optimization is used, according to which reduction steps and extension steps into unit clauses do not increase the current inference value. This implies that $\Delta i = 0$ if no extension steps in non-unit clauses have been performed on subgoals of the current clause.

5. The SETHEO System

5.1 SETHEO System Overview

The SETHEO system consists of a set of (individual) programs, as shown in Figure 5.1. Data are exchanged between the modules using files. Parameters for each program are given in the command line.

Figure 5.1: Parts of the SETHEO system

The indivual programs will be described in detail below. They are:

inwasm This program is the *compiler* for SETHEO. It takes a formula (as a set of clauses) or a logic program as its input. Details of the input language (LOP) and its syntax are described in detail in Section 6.2. **Inwasm** performs several preprocessing steps which try to reduce the search space spanned by the formula. Typical preprocessing steps are the generation of syntactical constraints or the elimination of pure literals. Then, binary code for the SETHEO Abstract Machine SAM are generated. Optionally, **inwasm** can output the preprocessed formula in LOP-syntax or SAM-assembler code. This option is helpful, if the preprocessed formula is to be modified by hand (or some other filter), before it should be compiled for the SAM. Details are discussed in Section 5.4.2.

sam This program is the interpreter of the SETHEO Abstract Machine SAM. After loading the code-file as prepared by **wasm**, the abstract machine is started with given parameters. If a proof could be found, the given time-limit has been exceeded, or upon request, statistical information is printed on the screen and

into a log-file. In case a proof could be found, the tableau with instantiated literals is generated and written into a file.

xptree This program is an X-based graphical viewer for tableaux, generated by **sam.** The tableau is displayed as a tree with literals (and additional information) as its nodes. Scrolling, selection and hiding of subtableaux can be accomplished using the mouse.

setheo This script represents the top-level script of the SETHEO system. Given a formula in LOP-notation, the compiler (inwasm) and the prover (sam) are activated automatically, using *default parameters.*

Around these basic programs, additional modules are located which use the basic programs. They increase the functionality of the SETHEO system and facilitates its usage.

Not all of the modules are currently supported, nor are described in this manual. (Those described in the following are marked by a †.) If you are interested in one of the modules, please contact the SETHEO-team.

New modules are likely to be added in further distribution versions of SETHEO. Therefore, modules which are new for the current version of SETHEO are marked clearly. The modules are listed in alphabetical order.

delta is a preprocessor which takes a formula in LOP syntax, and produces one in the same syntax. **Delta** generates unit-lemmata in a bottom-up way which are added to the original formula. In many cases, this combination of bottom-up preprocessing (using **delta**) and subsequent top-down processing by the SAM results in a dramatic increase of efficiency. **Delta** uses **inwasm, wasm** and **sam.**

plop This program[1] converts formulae in first order notation (with quantifiers and standard (infix-) operators (e.g., $\wedge, \vee$) into a set of clauses.

rctheo is a parallel theorem prover, based on SETHEO [?]. It uses random competition as the underlying model and is running on a network of workstations. **Rctheo** is available upon request.

sicotheo is a prototypical implementation for exploiting parameter competition with SETHEO. This implementation is based on **pmake** and is available upon request.

sptheo exploits parallelism by Static Partinitioning. Implemented upon PVM, it is running on networks of UNIX workstations [?].

wasm The assembler **wasm** converts assembler instructions (as normally produced by **inwasm** and converts them into a binary (currently actually hexadecimal) representation which can be directly read into the SETHEO Abstract Machine. Symbolic labels and constants are resolved. Furthermore, the code is somewhat optimized to reduce the size of the produced code file.

[1]This program is distributed as an "as is" version without support. For details see Section 5.4.1.

xvdelta This module is a primitve graphical user interface (GUI) for **delta** preprocessor. It is based on XVIEW toolkit of X. **Xvdelta** facilitates the selection of parameters of **delta** and to control its operation.

xvtheo This module is a primitve graphical user interface (GUI) for SETHEO. It is based on XVIEW toolkit of X. **Xvtheo** allows to edit a formula, to select appropriate parameters (using pull-down menues), to start the prover and to view the results. **Xvtheo** calls all the basic progams.

5.2 Filename Conventions

All files used within the SETHEO system carry specific file name extensions. These extensions are mandatory and need *not* be given when a SETHEO command (see Chapter 2, 4) is issued. In the following we list all possible file name extensions and describe their contents. Formal definition of the syntax is given in Chapter 6.

file.**pl1** Files with this extension contain formulae in first order predicate logic in mathematical syntax. Formulae can contain the standard operators (like $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$) and existential and universal quantifiers. An ASCII representation of the quantors is given, e.g. `forall` for $\forall$. A detailed description and formal definition is given in Section 6.1.

file.**lop** Files with this extension contain formulae and/or logic programs written in LOP syntax. This syntax comprises the default for the SETHEO system. A detailed description and the syntax definition is given in Section 6.2. Built-in predicates which can be used for logic programming are listed in Chapter 7. Files with the extension **.lop** comprise the input of the compiler **inwasm**.

*file*_**pp.lop** Files with this extension are produced by the compiler **inwasm**, when it is invoked with the command-line option `-lop`. This file contains the formula after preprocessing. Clauses are fanned into contra-positives, constraints are added to the clauses (if selected), and the weak-unification graph is given. Such a file can be read again by the compiler **inwasm**. Thus, manual (or automatic) modifications of the formula between preprocessing and the proof attempt can be accomplished.

file.**out.lop** Files with this extension are produced by the DELTA iterator, when given *file* as input. This file contains (in LOP syntax) the original formula (as found in *file*.**lop**) with the newly generated unit clauses appended to it.

file.**s** Files with this extension contain SAM assembler statements and are the input of the assembler **wasm**. Assembler files are generally generated by the compiler **inwasm**, when invoked with the option `-scode`. Nevertheless, assembler files are (to some extend) human-readable and can be modified (for prototypical implementation issues).

file.**hex** Such files are generated by the compiler **inwasm** (per default) or the assembler **wasm** and contain the binary (i.e. hexadecimal) representation of SAM-instructions and SAM's symbol table. **.hex** files are directly loaded by the SETHEO Abstract Machine SAM (command `sam`).

file.**tree** The prover `sam` generates files with that extension, if a proof has been found or upon request. Such files contain one or more Model Elimination Tableaux which then can be displayed graphically by the proof tree viewer **xptree.** The syntax of each tableau corresponds to a PROLOG term. Therefore, tableaux can be read in by a PROLOG system or (after adding predicate symbols to it) by the compiler **inwasm.** The syntax of the tree-files are described in Section 6.5.

file.**log** This file is generated by the prover `sam`. It contains useful information about the current run and comprises a copy of the data displayed on stdout. Besides command-line switches and a log of the iterative deepening, this file contains statistical information which is printed when the SAM stops (successfully or not).

Two further types of files are used within the SETHEO-system which do not carry specific extensions. The SAM-output file is opened by the built-in predicate `$tell/1` and contains all subsequent output as produced by other built-ins (e.g. `$write/1` and `$dumplemma/0`). For details see Chapter 7.

The **xptree** operator file contains a translation table of all operators together with their binding power. Its syntax is defined in Section 6.5.2.

5.3 The Environment

The shell scripts mentioned in Section 5.1 refer to the environment variable `SETHEOHOME` which must be defined in the environment settings. `SETHEOHOME` is the directory where the binaries of the basic programs **plop**, **inwasm**, **wasm**, **sam** and **xptree** reside, e.g. `/home/setheo/bin`. This variable is set accordingly, when the SETHEO system is installed (see Section 3.3). You can define the environment variable either by typing

```
setenv SETHEOHOME /home/setheo/bin
```

or by adding the line

```
setenv SETHEOHOME /home/setheo/bin
```

to your .cshrc/.tcshrc file or by adding the lines

```
SETHEOHOME = /home/setheo/bin
export SETHEOHOME
```

to your .login or bourne-/ ? /... shell startup.

5.4 The Basic Programs

5.4.1 plop

Plop[2] takes formulae of first order predicate logic in standard notation and converts them into LOP clausal sets preserving the property of unsatisfiability. Obvious tautological clauses and redundant literals are removed. Both, the first-order-language

[2]This program is provided on an "as is" basis, and is not maintained any more.

syntax and the LOP syntax are described in a later section[3]. **Plop** can be invoked by the following command line:

```
plop [-necho][-nopti][-sko][-neg][-baum][-debu][-defgro] file[.pl1]
```

The only parameter which has to be given is the name of the input file. It must have the extension **.pl1**. The generated output file has the extension **.lop** which can be handled by **inwasm**.

The other parameters are optional and have the following meaning:

–necho suppresses the screen output. Otherwise the input and output formula is displayed.

–nopti suppresses the removal of tautologies and redundancies.

–sko produces output clauses in skolemized normal form, i.e. clauses containing disjunction and negation symbols, but neither conditional nor conjunction symbols.

–neg negates the input formula, before doing the transformation. For many applications, when the formula is of the form $Axioms \rightarrow Theorem$, this switch must be used.

–baum displays statistics of the internal formula tree.

–debu turns on the verbose debugging mode of **plop**.

–defgro displays constraints concerning the size of some elements of the input formula. If a given example exceeds them, define-statements in the source code of **plop** are to be modified.

5.4.2 inwasm

Inwasm (**in**put for **w**arren **as**sem**b**ler) takes clauses in LOP notation and compiles them into SAM assembler code. The output may then be processed by **wasm**. For the syntax of the input language LOP, see the description in Section 6.2. **Inwasm** distinguishes between declarative clauses (axioms) and procedural clauses (rules). In the Non–Horn case, all declarative clauses are fanned into contrapositives. Also code to perform reduction steps is added in this case.

The Synopsis of **inwasm** is the following:

```
inwasm  [-[no]purity] [-[no]reduct] [-[no]fan] [-[no|x]sgreord]
        [-[no]clreord] [-randreord] [-all] [-taut] [-reg] [-eqpred]
        [-[r]subs] [-overlap[x] [number]] [-cons] [-ocons[x]]
        [-[r]linksubs] [-foldup[x]] [-folddown[x]] [-notree]
        [-searchtree] [-interactive] [-verbose [number]] [-lop]
file
```

The inputfile is the only parameter which has to be given. It must be a file with extension **.lop**. The output of **inwasm** is either a file with extension **_pp.lop** (see

[3]See Section 6.1, 6.2

option **–lop**) or with extension **.s** (default). **file_pp.lop** is again an input file for **inwasm** and **file.s** is an input file for **wasm** (see Section 5.4.3).

All optional parameters are listed below:

In the **inwasm** some default settings are integrated. For example a simple reordering of input clauses (short before long) and their subgoals (according to the first fail principle) is set by default. Switching off (or enforcing) the default settings can be done with the following options:

–[no]purity Suppress or enforce *purity reduction*. Purity reduction means that clauses are deleted, if they contain at least one subgoal which is not part of another clause. Purity is performed by default.

–[no]reduct Suppress or enforce code generation for *reduction* steps. Reduction steps guarantee completeness in the non–horn case.

–[no]fan Suppress or enforce *fanning* of declarative clauses, if the formula contains at least one non–horn clause.

–nosgreord Suppress *subgoal reordering* of declarative clauses.

–sgreord Enforce subgoal reordering of procedural clauses.

–[no]clreord Suppress or enforce *clause reordering*. Procedural clauses and declarative clauses are not distinguished.

The default settings of **inwasm** are **–purity –reduct –fan –sgreord –clreord.** Constraints can be computed in order to avoid redundant substitutions. These are the options for enabling constraints[4]:

–taut Instructions to add *tautology constraints* are added.

–reg Instructions for full *regularity* check via *constraints* are inserted.

–eqpred Instructions to detect *identical–predecessor–fails* are added. **–eqpred** implements a restricted form of **–reg.**

–subs Instructions to activate *unit subsumption constraints* are inserted.

–rsubs Instructions to activate *unit subsumption constraints* are inserted, but facts are ignored as subsuming clauses.

–overlap [number] Common initial segments of contrapositives of declarative clauses are detected and extracted. If possible, *overlap constraints* are generated. *number* specifies the increasing complexity [0..10] of these overlaps, the default is $number = 2$.

–overlapx [number] Like **–overlap**, but in this case, overlaps are also computed for the procedural clauses. Again, the default is $number = 2$.

[4]For tautology, subsumption and regularity constraints see also Section 5.4.4.

–cons Regularity, tautology and unit subsumption constraints are computed and inserted into the code file. Facts are ignored as subsuming clauses. **–cons** is the same as **–reg –taut –rsubs.**

–ocons [number] Regularity, tautology, unit subsumption and overlap constraints are computed and inserted into the code file. Facts are ignored as subsuming clauses. Overlaps beetween procedural clauses are not computed. *number* specifies the increasing complexity [0..10] of these overlaps, the default is *number* = 2. **–ocons [number]** is the same as **–reg –taut –rsubs –overlap [number].**

–oconsx [number] Regularity, tautology, unit subsumption and overlap constraints are computed and inserted into the code file. Facts are ignored as subsuming clauses. Overlaps beetween procedural clauses are computed. *number* specifies the increasing complexity [0..10] of these overlaps, the default is *number* = 2. **–oconsx [number]** is the same as **–reg –taut –rsubs –overlapx [number].**

As an internal representation of the input formula a weak–unification connection graph is generated. This connection graph can be further preprocessed to delete unnecessary links. The options for link deletion are the following:

–linksubs [number] Tautological and subsumable links are deleted. At most *number* * 1000 links are tested. Select the number from [1..5]. The default is *number* = 6.

–rlinksubs [number] Tautological and subsumable links are deleted, but links into facts are ignored. Again, at most *number* * 1000 links are tested. Select the number from [1..5]. The default is *number* = 6.

The following are additional inference rules for the SAM. They can be enabled by setting the concerning **inwasm** options:

–foldup Insert code to factorize subgoals with previous ones.

–foldupx Insert code to use previously solved subgoals for pruning purposes in an extended regularity check.

–folddown Insert code to factorize subgoals with later ones.

–folddownx Insert code to use later subgoals for pruning purposes in an extended regularity check.

Remark 1 *Folding up and folding down are not compatible with each other. The extended regularity checks are not complete in combination with antilemmata*[5]**.**

Miscellaneous options:

–randreord Enables random reordering of or–branches (see also **rctheo** in Section 5.1).

–all With this option all possible proofs within the current bounds are enumerated. For each proof SETHEO fails and adds the new proof–tree to the **.tree** file.

[5] See Section 5.4.4.

Before using **xptree** the file must be splitted. The option is especially useful in combination with write–statements to display answer substituitions of query variables.

–notree Writing the proof tree into a file is disabled.

–searchtree Instructions are inserted to write parts of the search tree into **file.ortree.**

–interactive Instructions are inserted to run the SAM in an interactive mode.

–verbose [number] Achieves varying verbosity specified with *number*. According to *number*, you get some intermediate preprocessing output. The various possibilities are printed onto the screen when you ommit the *number*.

–lop Writes preprocessing output into **file_pp.lop**. No file with extension **.s** is generated. Since **file_pp.lop** itself is a LOP file it can be used as an input file for **inwasm**. To avoid multiple fanning all clauses in **file_pp.lop** are procedural clauses.

5.4.3 wasm

Wasm takes the output file of the **inwasm** and generates the SAM assembler code which can be interpreted by the SAM. This is the synopsis of **wasm**:

```
wasm [-verbose][-[no]opt] file
```

The input file has to be a file with extension **.s.** From this, a file with extension **.hex** is generated.

The parameters for controlling verbose mode and code optimization are optional:

–verbose Turn on verbose mode.

–opt Code optimization is done. The connection graph representation is optimized to use less space.

–noopt Code optimization is switched off.

5.4.4 sam

The SAM (= **S**ETHEO **A**bstact **M**achine) is a theorem prover for full first order logic. It is based on the connection tableau calculus and runs on many different machines. The implementation of the SAM is based on the *Warren Abstract Machine (WAM).*

The synopsis of the SAM is the following.

```
sam   [-d number] [-i number] [-cdd number] [-wd number]
      [-wd1 number] [-wd2 number] [-wd3 number] [-wd4 number]
      [-wd4flag] [-tc number] [-tcd number] [-tcd1 number]
      [-tcd2 number] [-sig number] [-sigd number] [-sigd1 number]
      [-sigd2 number] [-fvars number] [-sgs number] [-dr [number]]
      [-ir [number]] [-cddr [number]] [-wdr [number]] [-[no]reg]
      [-[no]st] [-anl [number]] [-noanl] [-hornanl [number]]
      [-nhornanl [number]] [-[no]cons] [-dynsgreord [number]]
      [-singlealt [number]] [-multialt [number]] [-spreadreducts]
      [-forcegr] [-forcedl] [-forcefvar] [-forceps] [-[no]lookahead]
      [-eq] [-altswitch number] [-shortcl number] [-longcl number] [-code number]
      [-stack number] [-cstack number] [-heap number] [-trail number] [-symbtab num
      [-seed number] [-v[erbose] [number]] [-realtime number] [-cputime number]
      [-alltrees] [-printlemmata] [-batch] [-debug]
file
```

The filename must have the extension **.hex**. During and after the computation statistical information is generated. This information is printed on the screen as well as in a file with the extension **.log**. In case a proof is found, a proof tree is generated. The proof tree file has the extension **.tree** and can be displayed using **xptree**. The following parameters are optional.

The connection tableau calculus is sound and complete for full first order logic, if the tableau branches may contain inifinitly many nodes. To avoid an infinite search space *fixed bounds* can be set: The depth of the tableau, the number of inferences, the inference resources for each branch, the number of free variables, the termcomplexity and the number of open subgoals can be restricted. But note that completeness is not ensured when using these restrictions[6]. These are the bounds for a finite search space.

–d number *Depth bound*. The (A-literal) depth of the tableaux to be enumerated is less than *number* or equal to *number*.

–i number *Inference bound*. The maximum number of leaf nodes of the tableaux to be enumerated is less than *number* or equal to *number*.

–cdd number *Clause dependent depth bound*. Something in between the **–i** and the **–d** option; in each inference with a clause of length n, the resources for each new leaf node are determined by the resources of their parent minus n. The root node starts with *number* inferences.

–wd number The *weighted depth bound* permits to combine the above bounds. The bound is computed by using four parameters $wd_1, \ldots, wd_4$ described below. The maximum value of this bound is determined by *number*.

When entering a clause, its weighted depth is modified depending on the number of subgoals of the clause (according to wd_4); this value is divided into two

[6]The SAM distinguishes between *bound failure* and *total failure*: *Bound failure* means that this proof might have failed due to the chosen bounds and *total failure* means that this problem can never succeed.

parts, a *free* part and an *adaptive* part (according to wd_1, wd_2); when a subgoal is selected, the adaptive resource depends on the inferences performed since the clause has been entered (according to wd_1). In the special case, where all parameters $wd_1, \ldots, wd_4$ are zero, the bound behaves like the depth bound.

The following options are for setting the parameters $wd_1, \ldots, wd_4$. Let n_{sgls} be the number of subgoals.

–wd1 number Set wd_1 to *number*. The free depth is incremented by

$$\frac{wd_1}{100}$$

–wd2 number Set wd_2 to *number*. The free depth is decremented by

$$\frac{n_{sgls} * wd_2}{100}$$

–wd3 number Set wd_3 to *number*. Let Δi be the number of inferences performed since entering the clause. The adaptive depth is divided by

$$1 + \frac{\Delta i * wd_3}{100}$$

–wd4 number Set wd_4 to *number*. If the **–wd4flag** is not set, then the weighted depth is decremented by

$$1 + \frac{n_{sgls} * wd_4}{100}$$

Otherwise the weighted depth is decremented by

$$\frac{100 + n_{sgls} * wd_4}{100 + wd_4}$$

–wd4flag This flag is important in combination with the **–wd4** option (see above). Per default, the **–wd4flag** is set.

–eq This flag controls the combination of the weighted depth bound with subgoal alternation and equality handling by STE-modification. For details see the parameters concerning subgoal alternation. Per default, **–eq** is switched off.

–tc number *Termcomplexity bound*. The maximum value of the sum of term complexities of open subgoals is set to *number*. The complexity of a term is the sum over the complexities of the contained subterms. The complexity of a free variable is 0 and the complexity of a constant or of a function symbol is 1. Here the *dag-complexity* is computed: The complexity of repeatedly occurring subterms is 1.

–tcd number *Depth-dependent termcomplexity bound.* The maximum value of the sum of term complexities of open subgoals is set to a value depending on the permitted tableau depth.

The value of the depth-dependent termcomplexity bound *tcd* is computed according to parameters tc_1, tc_2. Let d be the permitted tableau depth.

$$tcd = \frac{tc_1}{100} + \frac{d * tc_2}{100}$$

Per default, $tc_1 = 1000$ and $tc_2 = 100$. The following options are for setting the parameters tc_1, tc_2.

–tcd1 number Set tc_1 to *number*.

–tcd2 number Set tc_2 to *number*.

–sig number *Signature bound.* The maximum number of different signature symbols (i.e., predicate, function and constant symbols) is set to *number*.

–sigd number *Depth-dependent signature bound.* The maximum number of different signature symbols is set to a value depending on the permitted tableau depth.

The value of the depth-dependent signature bound *sigd* is computed according to parameters sig_1, sig_2. Let d be the permitted tableau depth.

$$sigd = \frac{sig_1}{100} + \frac{d * sig_2}{100}$$

Per default, $sig_1 = 150$ and $sig_2 = 200$. The following options are for setting the parameters sig_1, sig_2.

–sigd1 number Set sig_1 to *number*.

–sigd2 number Set sig_2 to *number*.

–fvars number *Variable bound.* Only *number* free variables are allowed at the same time. This option is not complete in combination with antilemmata constraints (see below).

–sgs number *Subgoals bound.* Only *number* open subgoals are allowed at the same time.

A finite search space without loosing completeness is provided by the following *iterative bounds*: An initial bound is increased until either a proof has been found or the problem has failed totally. Iterative search can be done on the (weighted) tableau depth, on the number of inferences or on the inference resources of each branch. These are the options for iterative search.

–dr [number] Use iterative-deepening search with depth bounds. The depth bound increment after each cycle is given by *number*. The default increment is 1 and the start depth bound is 2.

–ir [number] Use iterative-deepening search with inference bounds. The bound increment after each cycle is given by *number*. The default increment is 3 and the start inference bound is 5.

–cddr [number] Use iterative-deepening search with clause dependent depth bounds. The bound increment after each cycle is given by *number*. The default increment is 1 and the start clause dependent depth bound is 5.

–wdr [number] Use iterative-deepening search with weighted depth bounds. The depth bound increment after each cycle is given by *number*. The default increment is 1 and the start depth bound is 2. See also the parameters for weighted depth bound search above.

Remark 2 *From the recursive bounds* **–dr**, **–ir**, **–cddr**, **–wdr** *only one option can be selected. For the fixed bounds we can have as many as we want. Recursive and fixed bounds can be combined.*

Furthermore, the search space can be restricted by the use of *constraints*. The constraint choices are the following.

–reg Use *regularity constraints* in order to avoid multiple occurrences of the same literal in a tableau path.

–noreg Deactivate regularity constraints.

–st Activate *subsumption* and *tautology constraints*.

–nost Deactivate subsumption and tautology constraints.

–anl [number] Use *antilemma constraints* in order to avoid repetitive solutions of the same subgoals. *number* characterizes the minimum search that must have been explored to generate an antilemma. The default is *number* = 25.

–noanl Deactivate generation of antilemma constraints.

–hornanl [number] Provides an optimized version of antilemmata which generates antilemma constraints only for *relevant* variables. This option is not complete.

–nhornanl [number] Similar to **–hornanl**. This option is complete, but the pruning effect is less than by using **–hornanl**.

–cons Use antilemma, regularity, subsumption and tautology constraints.

–nocons Deactivates antilemma, regularity, subsumption and tautology constraints.

Remark 3

1. *If you did not select the corresponding options in the preprocessing phase of* **inwasm** *the options* **–reg**, **–st** *have no effect and the* **–cons** *option will only turn on antilemma constraints. See also Section 5.4.2.*
2. *The* **–cons** *and* **–no...** *options can be combined to switch on all constraints except for one. For example* **–cons –noanl** *will activate all constraints exept for the antilemma constraints. But do not turn around the order of these options!* **–noanl –cons** *means: First switch off the antilemma constraints (even if they*

are not yet activated) and then switch on all constraints (including the antilemma constraints).

In finding proofs it is useful first to call the subgoals which are difficult to prove and try the easy ones later. The reason is to detect as early as possible, if a clause will lead to a fail. That is why *subgoal reordering* is a helpful method. A *static* subgoal reordering can already be done during the preprocessing phase of **inwasm** (see option **–sgreord** in Section 5.4.2). But during computation subgoals get instantiated and the initial order might not be optimal any longer. So a *dynamical* subgoal reordering becomes useful.

–dynsgreord [number] If the clause is declarative, a subgoal is chosen with respect to the strategy indicated by *number*. If the clause is procedural, no dynamical subgoal reordering is performed, i.e., the first open subgoal is taken (see also Remark 4). The default value for *number* is 1.

number = 1 The subgoal with the highest dag-complexity is chosen (The dag-complexity of a subgoal is the sum of the dag-complexities of its arguments.). If there is more than one subgoal with highest dag-complexity, the first of them is taken.

number = 2 If there are ground subgoals, only ground subgoals are considered in choosing. The subgoal with the highest dag-complexity is chosen. If there is more than one subgoal with highest dag-complexity, the one of them with the least free variables is chosen.

number = 3 The subgoal with the least free variables is chosen. If there is more than one subgoal with least free variables, the one of them with the highest dag-complexity is chosen.

number = 4 Like **–dynsgreord 1**, but do not switch to a subgoal with a different predicate symbol, according to the order of subgoals computed statically.

number = 5 Like **–dynsgreord 2**, but do not switch to a subgoal with a different predicate symbol, according to the order of subgoals computed statically.

number = 6 Like **–dynsgreord 3**, but do not switch to a subgoal with a different predicate symbol, according to the order of subgoals computed statically.

One common principle of standard backtracking search procedures is that the possible inference steps at a subgoal (the *choice point*) cannot be partially processed. More precisely, when backtracking to a choice point, another alternative in the *same* choice point must be tried. This methodology has search-theoretic weaknesses, as demonstrated in [Ibens and Letz, 1996]. With *subgoal alternation* a subgoal can be delayed, even if its choice point is not completely processed, and another subgoal may be selected. For procedural clauses, subgoal alternation is suppressed. The following options can be used to control subgoal alternation.

–singlealt [number] Invoke the *single-alternation* variant of subgoal alternation, i.e., move away from a subgoal at most once, namely after the reduction steps and the extension steps with unit clauses have been tried. The move is suppressed if the remaining depth resources are less than *number* (default: *number = 0*).

–multialt [number] Invoke the *multi-alternation* variant of subgoal alternation, i.e., a subgoal may be left whenever all its extension steps with clauses of a certain length have been processed. Leaving a subgoal is suppressed if the remaining depth resources are less than *number* (default: *number = 0*).

–spreadreducts Per default, when selection another subgoal, the one is selected which allows an inference step with adding the least subgoals to the tableau. Obviously, the number of added subgoals is zero for reduction steps and extension steps with unit clauses and n for extension steps using clauses with n subgoals. If **–spreadreducts** is switched on, not zero is associated to reduction steps but the least n available in the same choice point. Per default, **–spreadreducts** is switched off.

–forcegr Suppress subgoal alternation if the current subgoal is ground (default: switched off).

–forcedl Extend subgoal alternation from the use after backtracking to the use after successfull unifications, i.e., the length of the available alternatives is used as an additional criterion for subgoal selection (default: switched off).

–forcefvar Switch to a subgoal which has at least one variable in common with the current subgoal (default: switched off).

–forceps Switch to a subgoal which has the same predicate symbol as the current subgoal (default: switched off).

Remark 4 *If you are running the* SAM *on a file with the extension* **_pp** *(generated by inwasm with the* **–lop** *option), dynamic subgoal selection and subgoal alternation have no effect. This is because a file with the extension* **_pp** *contains only procedural clauses. See also option* **–lop** *in Section 5.4.2.*

–altswitch number This option allows to switch dynamically between two parameter selections for subgoal alternation. The possible parameter selections are the following:

0. no subgoal alternation
1. **–singledelay**
2. **–singledelay 2**
3. **–singledelay –forcedl**
4. **–singledelay –spreadreducts**
5. **–multidelay**
6. **–multidelay 2**

7. **–multidelay –forcedl**

8. **–multidelay –spreadreducts**

The switch is performed according to the length of the current clause: clauses of length $\leq$ *number* are viewed as short clauses, clauses of length $>$ *number* are viewed as long clauses. Per default, for short clauses the parameter selection 2 and for long clauses the parameter selection 7 are used. In order to change the these default setting, use the following commands:

–shortcl number For short clauses the parameter selection indicated by *number* is used.

–longcl number For long clauses the parameter selection indicated by *number* is used.

Remark 5 *Of course the* **–altswitch** *option can be combined with any of the above options for controlling subgoal alternation, e.g., the* **–forcegr** *option. But note that in this case the additional control option refers to both short clauses and long clauses.*

–[no]lookahead Subgoal alternation leads to processing several choice points simultaneously. This offers the possibility of computing the minimal number of subgoals that still will be added to the tableau and – since at each subgoal at least one inference step has to be aplied – the minimal number if inferences still needed for closing the tableau. This *look-ahead information* is used for search pruning when using the inference bound, the clause dependent depth bound, or a weighted-depth bound. Per default, **–lookahead** is switched on. Use **–nolookahead** to suppress the use of look-ahead information.

–eq For using look-ahead information in combination with the weighted-depth bound two different strategies have been developed. One of these is especially suitable when performing equality handling via STE-modification, and the other when performing axiomatic equality handling. Use **–eq** for equality handling via STE-modification. Per default, **–eq** is switched off.

Remark 6 *Subgoal alternation expects the alternatives in a choice point to be ordered by static clause selection So do not invoke subgoal alternation after having compiled an input file with* **inwasm –noclreord**. *This would lead to undesirable swichting orders and, in combination with the* **–lookahead** *option, to delaying proofs to higher iterative deepening levels.*

The SAM might stop with an *overflow error*. This means that the default sizes of the SAM's memory areas have been too small. You can enlarge the allocated memory with the following options.

–code number Set the maximal size of the *code area*[7] (default = 100 KBytes). The code area is the place to store the compiled input formula, i.e., the compiler output as generated by **wasm**.

[7]The size of the code area is different to the size of the actual formula.

–stack number Set the maximal size of the *stack* (default = 100 KBytes). The stack is the main working space of the SAM. For example, choicepoints with open alternatives are stored on the stack.

–cstack number Set the maximal size of the *constraint stack* (default = 100 KBytes). Constraints are not stored on the stack. They have their own memory area, the constraint stack.

–heap number Set the maximal size of the *heap* (default = 300 KBytes). Things in the code area can not be changed during computation. So non-ground terms have to be copied on the heap before they can be instantiated.

–trail number Set the maximal size of the *trail* (default = 200 KBytes). On the trail all changes are written that must be undone in case of backtracking.

–symbtab number Set the maximal size of the *symbol table* (default = 1000 entries). The symbol table contains all pedicate, function and constant symbols. Increasing the size of the symbol table means increasing the number of allowed symbols.

Remark 7 *The size of the code area and the size of the symbol table are set automatically to values appropriate for the given formula. Therefore, these options should be given in specific (e.g. debugging) situations only.*

These are miscellaneous options.

–seed number The random number generator is initialized with *number*. The random number generator is used, if we have set the compiler option **inwasm** *–randreord* to select the clauses at each orbranch in an arbirtrary permutation.

–v[erbose][number] Turn on verbose mode (for debugging). Different options for display are bitwise OR-ed.

–realtime number This option limits the run of the SAM abstract machine to at most number seconds (wall clock). If after that time no proof has been found, the SAM is aborted with `REALTIME FAILURE`. Please note that the accuracy of the alarm clock is 1 second.

–cputime number If this option is used, the SAM is aborted (with `CPUTIME FAILURE`) after the SAM has consumed *number* seconds CPU (user-) time. Due to the timer's accuracy of 1 second, the absolute run-time may differ up to 1 second from *number*.

–alltrees Per default, only the first generated proof-tree is written into the **.tree** file. This option causes *all* generated proof-trees to be written into the **.tree** file.

–printlemmata If the **–printlemmata** option is set, all generated unit lemmata are put out. The output is either written on the screen or into an output file which has to be specified by using the built-in $tell/1 (see Section 7.4.2).

–batch This option disables the interactive monitor mode after ⟨**Ctrl**⟩**–c** is pressed. This option should be used, if SETHEO is used within a script or a system. The SAM is aborted with the message `INTERRUPT FAILURE`.

–debug number This option can be used to obtain debug information. *number* is used to control the generation of debug information. The default value for *number* is zero which means that no debug information is given.

During the proof, the SAM can be interrupted by pressing ⟨**Ctrl**⟩**–c**. Then after entering **help** or ⟨**?**⟩ the following commands can be entered.

stat Display statistics.

r[eg] Display registers.

stack Display stack.

trail Display trail.

heap Display heap.

tree Display proof-tree.

links Display links.

choice Display current choices.

todo Display what is to be done.

cont Continue.

re Reproof.

help Help.

? Help.

! Shell escape.

quit Quit SETHEO.

5.4.5 xptree

Xptree is an X–window based interactive user interface for displaying proof trees generated by SETHEO resp. by the SAM. The user can manipulate the outline of the tree by showing interesting information and hiding information, which is useless to him. All ways of manipulation are mouse driven and so **xptree** is easy to use. The program is based on the *Athena Widgets programming library*, and can be operated with an arbitrary window manager. Predicates and functions can be displayed in prefix, infix, or postfix notation in order to enhance the readability of the proof tree. The representation and binding power of these operators can be defined by the user. The synopsis of **xptree** is the following:

```
xptree [-td number][-horiz][-vert][-ctab opfile] [file[.tree]]
```

The parameters for **xptree**, even the filename, are optional. The name of the tree file, which is displayed, must have the extension **.tree**. The extension can be omitted. If the whole file name is omitted, input is read from standard input. The other optional parameters are described below:

–td number Set the shown term depth. Terms, cascaded deeper then the specified number of levels are replaced by '...'. *number* must be greater than 0. The original term can be shown by pressing the middle mouse button (see also the description of the mouse buttons).

–horiz Display the tree horizontally. Can be changed interactively.

–vert Display the tree vertically. Can be changed interactively.

–ctab opfile This option loads *opfile* which contains optional definitions of infix, prefix, and postfix operators and their representation. The format of this file is described in Section 6.5.2.

Using the **–ctab** option (see above for a decription of this option) a file containing *operator definitions* can be loaded. The format of the file, describing the type of the operators (infix, prefix, postfix), their binding power and their print–names is as follows: Each line of the file contains one definition. Blank lines or comment lines are not permitted. A definition consists of four fields, separated by a colon (':'):

The parse–name is the name of the predicate or function symbol as it appears in the tree file produced by the SAM. It starts with a lower case letter and may contain lower and uppercase letters, digits, and the underscore '_'. Blanks and other special characters are not allowed.

The print–name is the string which is displayed whenever a term with the given operator is displayed. Print–names may contain arbitrary characters (except the colon, which has to be preceeded by a back–slash). In particular, a print–name can contain blanks to enhance the readability of a term.

The binding power is an integer, defining the binding power of the operator.

The type is a character, indicating the type of the operator: 'i' or 'I' for infix, 'p' or 'P' for prefix and 'q' or 'Q' for postfix.

Predefined operators are '~' and '[' as prefix operators with a binding power of 50. All entries in the given file are processed in reverse order. Therefore, the definitions of '~' and '[' can be changed by the user. The following example shows the format of the entries into the file:

```
equal  :  =   :  100  :  i
 in    :  in  :  100  :  i
ispre  :  [=  :  100  :  i
pair   :      :   95  :  p
 add   :  +   :  150  :  i
 mul   :  *   :  200  :  i
```

When the tree is displayed, the window can be distinguished into three areas: the menu bar, the panner and the tree itself. In each area the user can perform different actions: execute commands with the menu bar, move the tree in the window and change display of nodes. The second action is only available, if the tree is larger than the window. It is reached by moving the mouse in the panner area while the left button is pressed.

By using the menu bar, the following commands may be executed:

Quit Leave **xptree**. Key shortcut: **q**.

View/Layout Horizontal Display the tree with the root to the left. Key shortcut: ⟨**Ctrl**⟩–**c**.

View/Layout Vertical Display the tree with the root at the top. Key shortcut: ⟨**Ctrl**⟩–**v**.

Tree/Select All Show the additional information for every node.

Tree/Unselect All Hide the additional information for every node.

Tree/Select Children Show the additional information for each node, which has a currently selected parent.

Tree/Select Descendants Show the additional information for each node, which has a currently selected ancestor.

Tree/Set Term Depth Change the term depth dynamically (see also option **–td**). This option is not supported by now.

About Show the current version number.

The style of the tree can be changed by clicking the mouse buttons while the mouse pointer is on a node of the tree. The node must be highlighted, before any action is available. The actions are described below:

Left Mouse Button Toggles the display of additional information for the node.

Middle Mouse Button Toggles the display of term: brief vs. full display. The term depth option **–td** must be set.

Right Mouse Button Collapses the whole subtree of the node and the node itself to a small triangle. So the size of the displayed tree may be shrunk immensely.

Any mouse button, if the node is a triangle, expands the hidden subtree.

5.5 Additional Modules

The additional modules are shell scripts which call the basic programs. The environment variable `SETHEOHOME` should indicate the place where **plop**, **inwasm**, **wasm** and SAM reside[8]. If this variable is not defined, a default value is taken.

5.5.1 clop

Clop combines the SETHEO compiler **inwasm** with the assembler **wasm**. Given a file, first **inwasm** is invoked. All of the given parameters are passed to the **inwasm**. If the compilation phase terminated successfully, the assembler **wasm** is called with default parameters. The following is the synopsis of **clop**:

`clop` [`-par`$_1$]...[`-par`$_n$] `file`

[8] See Section 5.3.

The input file for **clop** has to be given. This must be a file with extension **.lop**. **Clop** will produce a temporary file with extension **.s** and generate the output file with extension **.hex**. The temporary file is not removed automatically. The **.hex** file can be used as an input for the SAM.

Clop returns 0 in case of successful compilation. Otherwise the non–zero return value of the module in which the error occurred, is returned.

5.5.2 setheo

Setheo calls the whole SETHEO theorem prover to prove a formula contained in a file *file.lop* using default options. **Setheo** in turn calls **inwasm**, **wasm** and SAM.

This is the synopsis:

```
setheo file
```

The name of the input file must have the extension **.lop**. During computation temporary files with the extensions **.s** and **.hex** are generated. These are not removed automatically. Two output files are generated. The output file with the extension **.tree** can be used as an input for **xptree** to visualize a found proof. The output file with the extension **.lop** contains the output of the prover SAM, the same as written onto the screen.

The following options are used (for details concerning these options see Sections 5.4.2, 5.4.3 and 5.4.4):

Compiler: Constraints are generated with **–cons**. The output code is optimized with **–opt**.

Prover: Iterative deepening over the depth of the tableau is performed (**–dr**). Constraints are generated and checked with **–cons**.

If the user wants to modify this default set of options, the user should refer to the commands **inwasm** and **wasm** for compilation and to the SAM for the prover itself.

5.5.3 delta

The DELTA-iterator is a preprocessing module which generates unit-lemmata from the input formula in a bottom up way. These unit-lemmata are then added to the input formula and then main top-down search for the proof is started. These additional unit-lemmata often abbreviate long (and difficult to find) sub-tableaux and thus often leads to a dramatic decrease of search-time for the overall system.

DELTA has a number of optional parameters and is invoked:

```
delta [-all] [-horn] [-nonhorn] [-noentry] [-debug] [-limit number] [-termsize numbe
    [-query] [-nosubs] [-large] [-cputime number] [-lastlevels number] [-termdepth n
    [-delta number] [-level number] [-pred predsymb arity] ... file
```

DELTA is a preprocessing tool for **Setheo**. It generates new single-literal clauses (facts) in a bottom-up fashion, using **Setheo**. A delta iteration method is used (see

below). The newly generated facts are appended to the original formula in the file *file*.lop to obtain the output file *file*.out.lop.

The orginal file *file*.lop remains unchanged. The number of newly generated clauses per level and the total amount of time needed for compilation and execution of SETHEO is printed at the end of the run.

The options for DELTA are:

-all With this option, unit-lemmata for all predicate symbols of the formula are generated. Furthermore, it is automatically determined, if the formula is Horn or Non-Horn (see **-[non]horn**).

-horn generate positive single-literal clauses only. (default)

-nonhorn positive and negative facts are generated. These are added as additional queries to the formula unless **-noentry** is given.

-noentry all newly generated negative facts are added as such to the formula.

-level *number* the given number of iterations are performed, or the given limit of newly generated clauses has been reached. (default: level = 1)

-limit *number* generate only less or equal than *number* new clauses. (default: limit = 999)

-termsize *number* generate (and keep) newly generated clauses only, if the total number of symbols in the terms of each clause is less than or equal than *number* symbols. Default: do not restrict the size.

-termdepth *number* generate (and keep) newly generated clauses only, if the maximal nesting level of symbols in the terms of each clause is less than or equal *number*. Default: do not restrict the depth of the terms.

-level *number* use the given number (default: 3) as the depth-bound for the SAM during the delta-iteration.

-pred name arity this option specifies the predicates, for which new clauses are to be generated. The arity of the predicate must be given. If the option **-all** is not set, at least one predictate must be specified.

-debug causes DELTA not to remove the temporary files after processing. Instead, their name is given.

For DELTA the following algorithm is used:

1. For each predicate given in the argument list or for all predicate symbols, special code is generated and added to the original formula ("special query"). Furthermore, code for the meta-level control is generated.
2. This formula is compiled and executed by the SAM.
3. The formula is entierly searched with a given low depth bound (which can be set using **-delta** *number*, default: 3). During the run, the newly generated single-literal clauses are stored in the Unit-lemma index. The unit-clauses are

kept only, if their term size and term depth do not viaolate the given bounds, and as long the limit of generated lemmas is reached.

4. If a stopping condition is met (number of levels or number of newly generated clauses is exceeded), the remaining clauses are added to the original formula and written into the file *file*.out.lop. Then DELTA prints the execution and compilation times and exits.

5. Otherwise, the search starts again (Step 3), whereby all unit-lemmata generated in the previous levels can be used during the search.

5.5.4 xvtheo

```
johann
```

5.5.5 xvdelta

```
johann
```

6. File Formats

6.1 The First Order Predicate Logic Syntax

In[1] this part we describe the syntax of the input language, which is accepted by the formula translator **plop.** A formula in that notation has to be in a file with the extension **.pl1.**

Because the grammar of first order logic is well-known it seems useless to repeat an inductive definition of the concept *"formula"*. We rather represent the special PLOP requirements i.e. the differences to the usual standard in giving some comments on the individual elements of the accepted language

Variables Variables are strings starting with a capital letter followed by arbitrary letters, digits, or underscore.

Constants Predicate-, function-, and object parameters are strings starting with a lower case letter followed by lower case letters, digits, or underscore. Predicate- and function arguments are to be included in round brackets. Multiple arguments are to be separated by a comma ",".

Propositional Connectives The symbols for the propositional connectives are:

(1) $\sim$ - not,
(2) `&` - and,
(3) `;` - or,
(4) `->` - if ... then,
(5) `<->` - iff.

The binding preference is given by this order: $1 > 2 > 3 > 4 > 5$. Equal connectives are left associative i.e., `a -> b -> c` is to be read as `((a -> b) -> c)` .

Quantifiers The quantifiers are represented by "`forall`" and "`exists`". These symbols must not occur as parts of other names like constants or variables. The scope of a quantifier extends over the following propositionally complete subformula and ends there. E.g. at the formula

```
forall X a -> b(X)
```

the scope of the quantifier does not include `b(X)`. To bind `X` by this quantifier one has to write as usual with parenthesis

```
forall X (a -> b(X)).
```

[1]This text is from the OLD SETHEO MANUAL.

Occurences of free variables are understood by PLOP as bound from the outside (universal closure).

Brackets Parenthesis are expressed by the usual *round brackets* only. *Square brackets* must not be used in the whole input formula .

Special Characters All characters with ASCII value < 32 are ignored and will be deleted during the transformation. The *space-character* can be omitted if the meaning is not affected and if there cannot result any ambiguities. E.g.

```
forallX
```

is recognized as a quantifier and a variable. In other cases a space character must be set, e.g.

```
forall X a(X)
```

cannot be written as

```
forall Xa(X).
```

Other characters as mathematical function symbols (especially in infix notation) are not recognized.

Comments Comments are enclosed in C–like convention in "`/*`" and "`*/`" and can extend over several lines. Nested comments are not allowed. All comments are removed during the conversion.

6.2 The LOP-Syntax

The general input language of the SETHEO-system is LOP (LOgic Programming Language) and is used to write formulas in clausal normal form (CNF) and to write logic programs, using SETHEO's procedural facilities. LOP formulas (or programs) are normally located in files with the extension **.lop** and are read by **inwasm** and the modules **setheo**, **clop** and **delta**. In this section, the exact definition of the syntax of a LOP formula or program is given in a BNF-like form[2].

The usage of white spaces (blanks, tabs and new-lines) is kept as flexible as possible. White spaces can be inserted or removed arbitrarily, unless otherwise noted. Comments are written in a C-like style surrounded by `/* */`. Nested comments are not allowed. A line starting with a `#` or a `%` is also treated as a comment.

LOP-files consist of a non-empty list of clauses[3] of different types (axioms, goals, queries and rules). Each clause is ended by a full stop '.'.

[2]This section is based from a definition of the LOP syntax by R. Letz

[3]In this manual, we only describe the syntax of pure LOP; the syntax of MPLOP, which further allows to specify modules, is not covered here.

```
lop-program   ::=
               clause '.'
               | clause '.' lop-program

clause        ::=
               axiom
               | goal
               | query
               | rule
```

Let's start with the types of clauses which are used for theorem-proving: axioms and goals.

```
axiom         ::=
               head '<-' tail constraints
               | head '<-' constraints

goal          ::=
               '<-' tail constraints
```

The head and tail of a clause are non-empty lists of literals; the `<-` separates the positive literals from the negative ones. `constraints` denotes (a possible empty) list of inequality constraints.

The logic-programming style clauses (query and rule) have a similar syntax which is close to PROLOG. The head of each rule can be just one literal only.

```
rule          ::=
               literal ':-' tail constraints
               | literal ':-' constraints

query         ::=
               '?-' tail constraints
```

In general, the following differences and similarities exist in the handling of these 4 types of clauses by SETHEO:

1. All literals in front of the separator have per default a *positive* sign, those after the separator (i.e. literals in the tail) have a negative one.
2. Each `goal` and `query` is used as a start clause (in the Model Elimination Calculus)[4].
3. If there exists at least one Non-Horn clause in the formula, each `axiom` and `goal` is transformed into contrapositives.
4. No `query` or `rule` is transformed into contrapositives.
5. An axiom with one literal only is syntactically and semantically the same as a rule with one literal ("fact").

[4]If a clause, consisting of negative literals only, is not to be used as a start clause, this clause can be transformed as described in Section 8.2.

6. Literals in the tail of a `axiom` and `goal` are subject to subgoal reordering (unless turned off with the **inwasm** switch `-nosgreord`), whereas those in a `query` or `rule` are not reordered.

The positive and negative parts of the clauses are lists of literals. A literal itself is an atom or its negation. Built-ins can occur in the tail of a clause only.

```
head          ::=
               literal
               | literal ';' head

tail          ::=
               tail-literal
               | tail-literal ',' tail

tail-literal  ::=
               literal
               | built-in

literal       ::=
               atom
               | '~' atom
```

In the following, we define the structure of an `atom`, which follows the usual lines of a definition of terms. Please note that additionally, *lists* of terms are allowed within terms. They are used in a PROLOG-like way.

As in PROLOG, constants start with a lower-case letter, variables with an upper-case letter or an underscore _. Anonymous variables are denoted like in PROLOG, by an _. Furthermore, short strings (quoted in ") are allowed. Please note that, however, a string `"plus"` and a constant `plus` are equal. This e.g. disallows a string `"*"`, since the asterisk is a reserved symbol. Special characters are expressed in a C-like manner: predefined are `\t` for tabulator, `\n` for newline and `\OOO` for a 3-digit octal value of the character's ASCII code. E.g. `\007` rings the bell. Numbers must be in the range of [-32768, 32767]. Additionally, the following consistency conditions must be fulfilled:

1. All predicate and function symbols can occur with one arity and as a constant of arity 0 only. This means that `f(a)` and `f` are allowed in one formula, but not `f(a)` and `f(a,b)`.

2. Predicate symbols may also occur as function symbols of the same arity (or used as a constant with arity 0), as long as the *first* occurrence of the symbol in that file is on the level of predicates. E.g. `<- p(a,b).  q(p)<-...` is allowed, but not the other way round. (Sorry)[5]

```
atom          ::=
               constant
               | constant '(' termlist ')'
```

[5]The best way to avoid this problem (in the inwasm compiler), is to preceed the formula with a line of the form $p(X_1, \ldots, X_n) \leftarrow \mathit{false}.$ for each predicate symbol p with arity n.

```
termlist        ::=
                 term
                 | term ',' termlist

term            ::=
                 constant
                 | constant '(' termlist ')'
                 | number
                 | variable
                 | global-variable
                 | list

constant        ::=
                 '[a-z][A-Za-z0-9_]*'
                 | '"[^"]*"'

number          ::=
                 '[0-9]+'
                 | '-[0-9]+'

variable        ::=
                 '[A-Z_][A-Za-z0-9_]*'

global-variable::=
                 '$[A-Z_][A-Za-z0-9_]*'

list            ::=
                 '[]'
                 | '[' term '|' term ']'
                 | '[' termlist ']'
```

Built-ins are either infix expressions or have a similar structure as atoms. They only can occur in the tail of a clause (i.e. they occur negatively). The names of the pre-fix built-ins start with a $. All built-ins are described in detail in Chapter 7.

```
built-in        ::=
                 built-in-pred
                 | built-in-pred '(' termlist ')'
                 | variable 'is' numexpr
                 | global-variable ':is' numexpr
                 | global-variable ':=' term
                 | term '=' term
                 | term '==' term
                 | term '=/=' term
                 | numexpr relop numexpr
```

```
numexpr        ::=
                number
                | '(' numexpr ')'
                | numexpr '+' numexpr
                | numexpr '-' numexpr
                | numexpr '*' numexpr
                | numexpr '/' numexpr

relop          ::=
                '<' | '>' | '<=' | '=='

buit-in-pred   ::=
                '$[a-z][A-Za-z0-9_]*'
```

Constraints are inequality constraints, i.e. lists of disequations of terms. Constraints can be considered as additional literals in the tail of a clause which are being checked permanently.

Please note that in the current version of SETHEO, constraints which have been added to the formula with a semantic meaning (e.g. `[X] =/= [0]`), are not compatible with SAM's built-in antilemmata mechanism. If you want to use such constraints, you must invoke SAM with the `-noanl` option or without `-anl` or `-cons`. All constraints generated by the compiler inwasm are pure syntactic constraints which do not influence soundness of the prover.

```
constraints    ::=            // empty
                | ':' constr-list

constr-list    ::=
                constraint
                | constraint ',' constr-list

constraint     ::=
                '[' ctermlist ']' '=/=' '[' ctermlist ']'

ctermlist      ::=
                cterm
                | cterm ',' ctermlist

cterm          ::=
                term
                | struct-variable
                | constant '(' ctermlist ')'

struct-variable::=
                '#[A-Z_][A-Za-z0-9_]*'
```

Structure variables `struct-variable` in constraints are treated as universally quanti-

fied variables (w.r.t. the clause in which they occur). A constraint `[X] =/= [f(#Y)]` is violated, if X is instantiated to a term with the top-level functor `f`, e.g. `f(a)`.

6.3 SAM Assembler Syntax

Assembler code for the SETHEO Abstract machine SAM is typically generated by the compiler **inwasm**. Per default, SAM-assembler code is kept in files with the extension **.s**. The assembler code can be processed by **wasm** in order to generate binary code for the SAM. The assembler language of SETHEO is defined in a very straight forward way. Each line of the input file can be empty, may contain a *directive*, a *label* or an *assembler instruction*.

```
assembler_code ::=
                line '\n'
                | line '\n' assembler_code

line           ::=
                LABEL ':'
                | '.' directive
                | statement

LABEL          ::= [A-Za-z][A-Za-z0-9_]*
```

Comments are C-like and enclosed in `/* ...*/`.

6.3.1 Directives

Each directive starts with a '.', followed by an identifier and arguments. Directives control the operation of the assembler and do not generate any code.

```
directive      ::=
                'include' STRING
                | 'equ' LABEL numexpr
                | 'dw' exprlist
                | 'ds' numexpr
                | 'start' numexpr
                | 'org' numexpr
                | 'symb' STRING ',' symbtype ',' numexpr
                | 'clause' numexpr ',' numexpr
                | 'red' numexpr
                | 'optim'
                | 'noopt'

symbtype       ::=
                'const' | 'var' | 'pred' | 'global' | 'gterm' | 'ngterm'

exprlist       ::=
                expr
```

```
                | exprlist ',' expr

exprlist        ::=
                numexpr
                | TAG numexpr

TAG             ::=
                'const' | 'cref' | 'eostr' | 'var' | 'gterm'
                | 'ngterm' | 'crterm' | 'cstrvar'
```

A numeric expression `numexpr` can be a number, a label or a sum or difference of numeric expressions. Labels need not be defined when they are used. However, if a label occurs in a sum or difference, it must be defined in order to yield a defined result. Both `+` and `-` are left associative.

```
numexpr         ::=
                LABEL
                | NUMBER
                | numexpr '+' numexpr
                | numexpr '-' numexpr

NUMBER          ::= [0-9][0-9]*
                | -[0-9][0-9]*
```

include includes the named file `STRING`. Nested includes are possible up to 8 levels.

equ With a directive of the form `.equ name expr`, a macro-name `name` is assigned the value of `expr`.

dw This directive allows to set one memory cell with a (tagged or untagged) word.

ds A directive `.ds number` reserves `number` consecutive memory cells. The value of the number must be known during the first pass.

start This directive allows the SAM to start at a given address (default: 0).

org Places the following code at a given memory address

symb This directive defines a symbol. Since the relative line number of this directive (w.r.t. the first `.symb`) line is used to determine the index into the symbol table, no entries must be added, exchanged or deleted.

clause This directive marks the beginning of a specific contrapositive or clause. This directive is used for debugging and readability purposes only.

red This directive marks code for performing reduction steps for a specific predicate symbol. This directive is used for debugging and readability purposes only.

optim, noopt These directives mark the begin and end of pieces of code to be optimized, respectively.

6.3.2 Statements

```
statement      ::=
                instruction0
                | instruction1 expr
                | instruction2 expr ',' expr
                | instruction3 expr ',' expr ',' expr

instruction0   ::=
                'stop' | 'told' | ...

instruction1   ::=
                'alloc' | 'isunifiable' | ...

instruction2   ::=
                'assign' | 'call' | ...

instruction3   ::=
                'eqpred' | 'porbranch' | ...

expr           ::=
                NUMBER
                | LABEL
```

6.4 SAM Machine Code Syntax

Machine code for the SETHEO Abstract Machine SAM is always located in files with the extension **.hex**. Each entry in this file occupies one line. A one-character identifier is used to select the appropriate type of data contained in the current line. Blank lines, comments or extra spaces are not allowed.

The following grammar shows the definition of the SAM Machine Code:

```
file           ::= lines

lines          ::= line
                | lines line

line           ::= ':' ident ':' address ':' data ':' string '\n'

ident          ::= 'C' | 'Y' | 'E' | 'S' | 'M' | 'N'


address        ::= [0-9A-F]{8}
data           ::= [0-9A-F]{8}
string         ::= [A-Za-z_0-9]        /* see Note 4 */
```

The following table shows the meaning of the fields, corresponding to the given identifier:

C	word in code area of SAM	SAM-code-address	contents	—
Y	symbol	type of symbol	arity	printname
S	set start-address	SAM-code-address	—	—
M	highest address	0	highest address	—
N	number of symbols	0	symbols	—

Notes:

1. A "highest memory address" (identifier: "M") directive must be placed prior to any code words ("C") in order to be effective. Otherwise, the size of the code-area of the SAM is determined by a default value (or the **sam** parameter -code).
2. The relative line number of a "symbol" directive determines the index of that symbol in the symbol table. Therefore, the order of these directives must not be changed and their number must not be increased or decreased.
3. Special characters in a symbol are encoded by: `\OOO` where `OOO` is the 3-digit octal value of the ASCII code of that character. E.g. `\007` is for `BELL`.
4. The maximal length of a symbol is limited to 42 on the generation side (**wasm**), and to 200 in the reader of the abstract machine. This limit includes special characters (see above). Each such character contributes 4 to the total length of the symbol.
5. all other directives may be intermixed and arbitrarily changed in order.
6. A file with machine code must at least contain one line.

6.5 Syntax Definition of a Proof Tree

6.5.1 The Tree File

Files with the extension *.tree* are generated by the SAM and contain Model Elimination Proofs, i.e., representations of closed tableaux. For each inference step, its kind (extension step, reduction step, factorization step), and additional information is kept. The literals of the clauses are stored with the variable-substitutions applied. The syntax of tree-file is defined in the following. All proof are stored as PROLOG terms, such that **inwasm** or a PROLOG program can read and process the proof.

```
proof ::= treelist '.'

treelist ::= '[' seq_of_trees ']'

seq_of_trees ::=
        '[' tree ']'
        | seq_of_trees ',' '[' tree ']'
```

One tree-file can actually contain more than one Model Elimination tableaux. Currently, however, **xptree** can only display one tree. In case more than one trees are given, obly the first one is displayed.

```
tree ::= literal { ',' '[' infnumber ',' infdescr ']' { ',' treelist } }
```

Each node of the tree contains a literal, together with (optional) information. Its is followed by the tree-representation of all its immediate successors. As an example consider the following clause: `p :- q,r,s.` Then, given the subgoal `~p`, the following tree structure is generated (the __ denote the addtional information:

```
[ ~p , [____],       // goal
  [ p ],             // head of the clause
  [ ~q , [____],
     [ <subtree> ]  // for subgoal '~q'
  ] ,
  [ ~r , [____],
     [ <subtree> ]  // for subgoal '~r'
  ] ,
  [ ~s , [____],
     [ <subtree> ]  // for subgoal '~s'
  ]
]
```

For each extension step, the list of subtrees `treelist` contains at least one element. If, however, a subgoal was closed by a reduction- or factorisation step, this list is empty.

The `literal` is defined in a similar way as in LOP (see Section ??). Here, however, no infix operations (except `=`) are allowed[6].

```
infnumber ::= [1-9][0-9]*

infdescr ::= 'ext' '(' number '.' number ',' number '.' number ')'
           | 'red' '(' number '.' number ','  number ')'
           | 'fac' '(' number '.' number ','  number ')'
           | 'built_in'
           | 'not yet touched'
```

The inference number is a unique sequence number for each node of the tree. The maximal inference number corresponds to the number of inferences in the proof. The item `infdescr` gives information about the current inference. The first pair of numbers identifies the literal, *from* which the inference was made ("subgoal"). A literal is always identified by its clause number (1st number) and the relative number of the literal in the clause. Note, that subgoal reordering does not affect the numbering.

For an extension step (`ext`), the second pair of numbers indicate, into which literal the extension step has been made. The second argument of the other inference steps

[6] Actually, all infix operations of LOP are handled with built-in predicates which do not show up in a proof tree.

denote the inference number of the node, into which a reduction (or factorization step) has been made.

`not yet touched` indicates a subgoal which has not been solved. This situation can only occur when a proof tree is generated during the serach of the SAM, using the built-in `ptree` (see Section ??). Finally, the token `buil_in` exists, but it is not used within the current version of SETHEO.

6.5.2 The Operator Translation Table

In the SETHEO system, all operators (except certain built-ins) are represented in prefix notation. The graphical output of SETHEO, using xptree, however, allows to display such operators in infix- or postfix notation, using definable print-names. A translation table must be present in a file and can be loaded using xptree's `-xtab` *file* option.

The file, describes the type of the operators (infix, prefix, postfix), their binding power and their print-names is as follows: Each line of the file contains one definition. Blank lines or comment lines are not permitted. A definition consists of four fields, separated by a colon (':'):

the parse-name
: is the name of the predicate or function symbol as it appears in the tree file produced by the SAM. It starts with a lower case letter and may contain lower and uppercase letters, digits, and the underscore '_'. Blanks and other special characters are not allowed.

the print-name
: is the string which is displayed whenever a term with the given operator is displayed. Print–names may contain arbitrary characters (except the colon, which has to be preceeded by a back–slash). In particular, a print–name can contain blanks to enhance the readability of a term.

binding power
: is an integer, defining the binding power of the operator.

type
: is a character, indicating the type of the operator: 'i' or 'I' for infix, 'p' or 'P' for prefix and 'q' or 'Q' for postfix.

```
convtab ::=
       ctabline
       | ctabline convtab

ctabline ::=
    pt_symbol^':'^char_sequence^':'^number^':'^optype^'\n'

pt_symbol ::=
symbol | '~' | '[' | ']'
```

```
char_sequence ::= [^:]+          /* string, must not contain ':' */

optype ::= 'I' | 'i' | 'p' | 'P' | 'q' | 'Q'
```

Predefined operators are '$\sim$' and '[' as prefix operators with a binding power of 50. All entries in the given file are processed in reverse order. Therefore, the definitions of '$\sim$' and '[' can be changed by the user. The following example shows the format of the entries into the file:

```
l: <= :100:i
equal: = :100:i
in: in :100:i
ispre: [= :100:i
less: < :100:i
filt:@:200:i
cons:*:200:i
gt: > :100:i

left-right associativity?
```

6.6 The SAM Logfile

While the SAM tries to find a proof, statistical information is generated. This information is printed on the screen as well as into the SAM logfile. The extension of the logfile is **.log**. An example of a SAM logfile is shown in Figure 6.1.

At top of the logfile the number of the SAM's version is printed, e.g.

SAM V3.3 Copyright TU Munich (December 22, 1995)

This is important because different SAM versions may produce different statistics. The next thing to be printed in the logfile are the options and the name of the problem file the SAM is called with, e.g.

Options : –dr –cons –dynsgreord 2 WOS4

Each sort of turned on constraints is listed as well[7]. The beginning of the search to the proof is indicated by

Start proving ...

If the SAM is called with an iterative bound (e.g. **–dr**), for each iteration step during computation the following information is printed.

1. The value of the increased bound. Since the SAM provides different iterative bounds, a short string before the value indicates which bound is increased.

   –d: Depth bound.

[7]This list refers only to the SAM's parameters and not to which constraints are really effecting the proof. I.e. some of the used constraints might have no effect if the corresponding options have not been turned on in the preprocessing phase of **inwasm** (see Remark 3 in Section 5.4.4).

–wd: Weighted depth bound.

–i: Inference bound.

–loci: Local inference bound.

2. The runtime (CPU–time) in seconds, introduced by the string **time**.
3. The total number of inferences, introduced by the string **inferences**.
4. The number of fails, introduced by the string **fails**.

When a proof is found,

******* SUCCESS *******

is printed into the logfile. Otherwise the reason for not finding a proof is given.

******* TOTAL FAILURE *******

means that the given problem is satisfiable. This message also indicates that this failure does not depend on SAM's invocation parameters.

******* BOUND FAILURE *******

indicates that the failure was caused by the given bounds. With different bounds set a proof might be found.

******* INTERRUPT FAILURE *******

is printed, if the SAM is running in the batch mode and the run is interrupted, e.g. by ⟨**Ctrl**⟩**–c**. Read Section 5.4.4 for an explanation of the **–batch** option.

******* REAL TIME FAILURE *******

is printed, if the SAM is running with limited real time and the given time–limit has been exhausted. Read Section 5.4.4 for an explanation of the **–realtime** option.

******* CPU TIME FAILURE *******

is printed, if the SAM is running with limited CPU time and the given limit has been exhausted. Read Section 5.4.4 for an explanation of the **–cputime** option.

******* E R R O R *******

is printed, if an internal SAM error occured. This indicates inconsistent changes to the source code or an illegal use of built–ins.

When the run is over all non–zero statistical counters are written to the logfile. The statistical counters are the following.

Number of inferences in proof: number of successful, not backtracked unifications

–E: number of successful, not backtracked extension steps

–R: number of successful, not backtracked reduction steps

–F: number of successful, not backtracked factorization steps

–L: number of successful, not backtracked unit–lemma uses (always zero in the current version)

Intermediate free variables: intermediate maximum number of simultaneously free variables

Intermediate term complexity: intermediate maximum term complexity (only displayed, if the termcomplexity bound is set)

Intermediate inferences: intermediate maximum number of inferences (i.e. successful, not backtracked unifications)

Intermediate open subgoals: intermediate maximum number of simultaneously open subgoals

Generated antilemmata: total number of generated antilemmata

Elapsed Time: elapsed time since last interrupt (only displayed, if the computation is interrupted, e.g. by ⟨**Ctrl**⟩**–c**)

Number of unifications: total number if tried unifications (including unsuccessful and backtracked unifications)

–E: number of tried extension steps

–R: number of tried reduction steps

–F: number of tried factorization steps

–L: number of tried unit–lemma uses (always zero in the current version)

Number of generated constraints: total number of constraints generated during computation (only displayed, if constraints have been generated)

–anl: number of antilemma constraints

–reg: number of regularity constraints

–ts: number of tautology or subsumption constraints

Number of fails: total number of fails (the following numbers are only displayed, if they are greater than zero)

–unification: number of unification fails

–depth bound: number of depth bound fails

–inference bound: number of inference bound fails

–variable bound: number of variable bound fails

–term complexity bound: number of termcomplexity bound fails

–open subgoals bound: number of subgoal bound fails

–local–inf bound: number of local inference bound fails

–constraints: total number of constraint fails

–anl: number of antilemma constraint fails

–reg: number of regularity constraint fails

–ts: number of tautology or subsumption constraint fails

Number of folding operations: total number of folding operations

–one level: number of folding operations one level up

–root: number of folding operations up to root, i.e. number of possible lemmata

Instructions executed: number of executed SAM instructions[8]

Abstract machine time: pure proving time[9] of the SAM program in seconds (i.e. runtime without time for reading the options and the formula and without time for displaying the statistics).

Overall time: total runtime[10] of the SAM program in seconds (i.e. time for reading the options and the formula, proving time, time for displaying the statistics)

Genlemma statistics concerning the generation of unit–lemma (only displayed, if the concerning numbers are greater than zero). For details see the **genlemma** built–in predicate in Section 7.6.1.

–entered: number of times, the built–in to generate unit–lemtata has been entered

–no-unitlemma: number of times, it had been tried to generate a non–unit lemma

–GEN/w.match: GEN indicates, how many more general unit–lemmata could be found in the index. **w.match** is the number of times, a really more general lemma could be found in the index. The ratio between **GEN/w.match** indicates, how good the filter provided with the path index really is. Note: Whenever a more general unit–lemma already exists in the index, the new lemma is not entered.

–INST/w.match: INST indicates, how many more instantiated unit–lemmata could be found in the index. **w.match** is the number of times, a really more instantiated lemma could be found in the index. The ratio between **INST/w.match** indicates, how good the filter provided with the path index really is. Note: Whenever a more instantiated unit–lemma is found, it will be removed from the index (i.e. it cannot be used any more). The new lemma will be stored in the index.

–stored: number of unit–lemmata actually stored in the index

–Nr. in Index: number of unit–lemmata which can be used by the prover

[8]Examples for SAM instructions are **calling a clause**, **calling a subgoal**, **backtracking**, **finishing a subproof**.

[9]These times are measured with a granularity of 1/60 seconds (depending on the UNIX system).

[10]These times are measured with a granularity of 1/60 seconds (depending on the UNIX system).

Uselemma statistics concerning the use of unit-lemmata (only displayed, if the concerning numbers are greater than zero). For details see the **uselemma** built–in predicate in Section 7.6.3.

–entered: number of times the built–in to use unit–lemmata has been called

–no: number of unsuccessful tries to use a lemma (In that case, the index does not return any unifiable unit–lemma.)

–Nr. pushed: number of times, unit–lemmata are activated to be used (i.e. appended to the current choice point)

: memory statistics (only displayed, if the SAM is running in verbose mode[11]), including

Pagesize

Marksize

Memory demanded

Memory freed

Memory remaining

Pages allocated

[11] Read Section 5.4.4 for an explanation of the **–verbose** option.

```
SAM V3.3 Copyright TU Munich (December 22, 1995)

Options : -dr -cons -dynsgreord 2 WOS4

using antilemma-constraints
using regularity-constraints
using tautology-constraints
using subsumption-constraints

Start proving...

-d:    2  time     < 0.01 sec  inferences =        5  fails =       29
-d:    3  time =     0.05 sec  inferences =      266  fails =     1176
-d:    4  time =     0.03 sec  inferences =      105  fails =      197

                  ******** SUCCESS ************

Number of inferences in proof      :       13
       - E/R/F/L                   :        9/       4/       0/       0
Intermediate free variables        :        3
Intermediate inferences            :       13
Intermediate open subgoals         :        5
Generated antilemmata              :        2
Number of unifications             :      376
    - E/R/F/L                      :      330/      39/       7/       0
Number of generated constraints    :       44
    - anl/reg/ts                   :        0/       6/      38
Number of fails                    :     1402
    - unification                  :      197
    - depth bound                  :     1165
    - constraints                  :       40
        - anl/reg/ts               :        2/       6/      32
Number of folding operations       :       12
    - one level                    :        5
    - root                         :        8
Instructions executed              :     2992
Abstract machine time (seconds)    :     0.08
Overall time          (seconds)    :     0.43
```

Figure 6.1: A SAM logfile.

7. Logic Programming

The basic Model Elimination Calculus and SETHEO's way of searching for a proof is quite similar to the way, a PROLOG program is executed. Therefore, we have added a variety of techniques to SETHEO which allows SETHEO to be used for logic-programming purposes as well as for pure automated theorem proving. In particular, these logic programming facilities allow to control the behavior of SETHEO. Thus, e.g., different search strategies, generation and usage of lemmata can be implemented easily (in a prototypical way).

In the current version of SETHEO, two major extensions are provided: global, backtrackable variables (see following Section **??**, and PROLOG-style built-in predicates.

Syntactically, a built-in predicate is an `ATOM` in the LOP-language, corresponding to the syntax definition in Section 6.2. Except for built-ins for arithmetic, all built-in predicates are of the form $\$p$ or $\$p(t_1, \ldots, t_n)$. The name of the predicate symbol always starts with a \$ sign in order to be able to distinguish syntactically between ordinary predicate symbols and built-ins. Such a built-in must alway occur in the *tail* of a clause. The complete list of built-ins is given in the following sections of this Chapter. For each built-in, we give its name, synopsis, a description, and one or more example demonstrating the usage of the built-in.

When using built-in predicates, several items must be taken into account:

- when using built-ins, SETHEO can become both unsound and incomplete.
- built-in predicates can have side-effects. Therefore, the order in which these predicates are executed are important. Since, per default, SETHEO reorders the subgoals of the clauses, it is advisable to turn off subgoal reordering (`-nosgreord` flag of **inwasm**), or to use procedural clauses (with a `:-` as separator.
- Although all built-ins can be used in combination with non-Horn clauses as well, no clear semantics can be defined in that case.
- The list, described in this manual, describes all predicates, available with SETHEO Version V3.3. Future versions will probably extend this list.
- When using built-ins and global variables, it might be advisable to turn off the search-space reduction techniques, since these might disable the execution of clauses with procedural side-effects only.

7.1 Global Variables

The concept of *destructive* and *multiple* assignment for variables is one of the basic concepts in a procedural programming language. In logic programs, however, a variable has quite a different meaning: it is a *logical variable* [**?**]. A logical variable

can be used and modified by the *unification* procedure only which may perform a *specialisation*.

This means that a variable which is in the beginning of the calculation *unbound* may be given several values during the proof, but each value is a specialisation of the previous ones (or the same). Otherwise unification is not possible. This behavior of logical variables does, of course, not allow multiple destructive assignments to a variable, e.g. `X:=5` and afterwards `X:=7`.

Another basic principle of procedural languages is the existence of *global variables* which are visible and valid from all parts of the program and in all incarnations of the procedures or functions. This is different in logic: in clausal form, a variable is valid only within *one* clause.

For a programmer who is using procedural languages most of the time, it would be very convenient to have the possibility of using (global) variables with multiple destructive assignment in a logic program. This inclusion also partly bridges the gap between procedural languages and logic languages, as the concept of global variables can be embedded cleanly into the logic language as shown below. This is in contrast to PROLOG which developed the concept of global data by introducing the *assert* and *retract* predicates. These constructs, however, have no clean and natural denotational semantics. Note that a multiple destructive assignment in a logical language must be *backtrackable* when the underlying execution model needs backtracking to find a solution (e.g. the SAM and PROLOG).

So we introduce global backtrackable variables (see also [?] for the theoretical background and their semantics). This concept has a clean denotational semantics, which has been introduced by R. Letz and will be shown below, and allows for a very efficient and easy integration into the SAM. The only draw-back of this construct is that we have to commit ourselves to a fixed order of evaluation of the subgoals (*literal selection function*), namely left-to-right. Only with this restriction we can make sure that the assignment `$X:=1` will be executed *prior* to `$X:=2` in the tail of a clause `...,$X:=1,...,$X:=2,...` which is equal to the intuitive meaning of (procedural) evaluation.

In LOP the concept of global variables and multiple destructive assignment is employed as a basis for almost all its non-logical and meta-programming constructs.

7.1.1 Syntax and Usage of Global Variables

In LOP global variables are denoted like variables, but their names start with a '$', e.g. `$X`, `$World` are global variables. Global variables can be used like ordinary variables in the terms of the predicates. Besides this, they may appear on the left hand side of the destructive assignment (`:=`). On the right hand side of an assignment any term may be present. In such a term the global variable which occurs on the left hand side of the assignment may be present as well, e.g. `$X:=f($X,a)` or `$X:=[a,$X]`.

7.1.2 Examples of the Usage of Global Variables

One short example shows how global variables, in our case `$X`, are backtracked.

```
                ← example.
example         ← $X := a, p.
p               ← $X := b, fail.   % must backtrack
p               ← write($X, ).     % print value of $X
```

When this program is executed, the variable `$X` is first set to the value `a`. After entering the first clause for `p`, the variable is set to `b` (destructive assignment). After that, a failure occurs which causes a backtracking step to be performed and the second clause can be tried, printing the current value of `$X`. If global variables were not backtrackable, the value after the last assignment, in our case `b` would be printed. This, however, is not correct. With the usage of *backtrackable* global variables as introduced above, the correct value, namely `a`, is printed.

7.1.3 Restictions for Global Variables

Global variables are in general not compatible with SETHEO's anti-lemma mechanism, as is shown with the following example:

```
?-p, q($X).
q(b).
p :- $X := a.
p :- $X := b.
```

This logic program first solves the subgoal `p`, whereby `$X` is assigned to `a`. Then the second subgoal is tried, but here the unification fails. During backtracking, the anti-lemma mechanism determines that `p` has been solved optimally (there are no substitutions for logical variables in `p`). Hence it is decided that there exist no more solutions, and thus the SAM returns with "total failure". If the generation of antilemmata is disabled, the above program works correctly.

7.2 Built-ins for Test, Arithmetic and Assignment

7.2.1 Destructive Assignment, :=

Synopsis: `$V := T`

Parameters:

`$V` is the name of a global variable.

`T` T is an arbitrary term which is gets (destructively) assigned to the global variable.

Low Level Name: `assign`

Result:

This statement always succeeds.

Description. The term `T` is not copied during the assignment. This means that later substitutions of variables present in `T` also alter the contents of the global variable.

Copying of the entire term can be accomplished using `$functor/3`.

Side-effects. Destructive term assignment to global variables.

Example.

```
p <- $G := [p_entered | $G],...
```

Here, we enter a symbol to a list which is kept in the global variable `$G`.

7.2.2 $unify/2,=

Synopsis: `$unify(T1,T2) , T1 = T2`

Parameters:

`T1,T2` two terms to be unified.

Low Level Name: `equ_unif`

Result:
This built-in succeeds, if both terms are unifiable.

Description. This built-in unfies to terms. The substiutions of the unification are applied to the terms `T1, T2`.

Note. If only a check is required, if two terms are unifiable, `$isunifiable/2` should be used.

Example.

```
p(X) <- $unify(X,f(g(a,b),c)).
```

produces the same result as

```
p(f(g(a,b),c)<-.
```

7.2.3 $isunifiable/2

Synopsis: `$isunifiable(T1,T2)`

Parameters:

`T1,T2` two terms to be checked.

Low Level Name: `is_unifiable`

Result:
This built-in succeeds, if both terms are unifiable.

Description. This built-in tests, if both terms are unifiable. The terms are not modified.

Example.

```
p(X) <- $isunifiable(X,f(g(Y,b),c)).
```

tests, if `X` is unifiable with `f(g(Y,b),c)`.

7.2.4 $isnotunifiable/2

Synopsis: `$isnotunifiable(T1,T2)`

Parameters:

`T1,T2` two terms to be checked.

Low Level Name: `is_notunifiable`

Result:
This built-in succeeds, if both terms are not unifiable.

Description. This built-in tests, if both terms are unifiable, and fails, if they are. The terms are not modified.

Example.

```
p(X) <- $isnotunifiable(X,f(g(Y,b),c)).
```

tests, if `X` is not unifiable with `f(g(Y,b),c)`.

7.2.5 $eq/2, ==

Synopsis: `$eq(T1,T2) , T1 == T2`

Parameters:

`T1,T2` terms to be checked for syntactical equality.

Low Level Name: `eq_built`

Result:
This built-in succeeds, if `T1` is syntactical equal to `T2`.

Description. This predicate checks two terms for syntactical equality. Two terms are syntactical equal, if

- both are the same symbolic constants (or strings)
- both are the *same* variable,

- the terms are of the form $f^1(t_1^1,\ldots,t_n^1)$ and $f^2(t_1^2,\ldots,t_n^2)$, and the function symbols f^1, f^2 and all pairs of subterms t_i^1, t_i^2 are syntactical equal.

Example.

```
p(X)<- $eq(f(a),X).
p(X)<- $eq(f(Y),X).
p(X)<- f(Y) == X.
```

The first clause succeeds, if X is instantiated to $f(a)$. The second clause, however, does *never* succeed, since Y only occurs only once in the second clause. The third clause is equivalent to the second one.

7.2.6 $neq/2, =/=

Synopsis: `$eq(T1,T2) , T1 =/= T2`

Parameters:

`T1,T2` terms to be checked for syntactical inequality.

Low Level Name: `neq_built`

Result:
This built-in fails, if `T1` is syntactical equal to `T2`.

Description. This predicate checks two terms for syntactical equality. For the definition of syntactical equality see `$eq/2`.

Example.

```
p(X)<- $neq(f(a),X).
p(X)<- $neq(f(Y),X).
p(X)<- f(Y) =/= X.
```

The first clause succeeds, unless X is instantiated to $f(a)$. The second and third clause, *always* succeed. The third clause is equivalent to the second one.

7.2.7 is

Synopsis: `V is NUMEXPR`

Parameters:

`V` (logical) variable

`NUMEXPR` term or numerical expression which must evaluate to a number when this statement is encountered.

Low Level Name: `sto`

Description. This statement evaluates the numerical value of `NUMEXPR` and unifies the result with the variable `V`. `is` succeeds, if `V` is an unbound variable or if `V` is instantiated to a number which is equal to the value of the numerical expression `NUMEXPR`. Otherwise a fail occurrs. If `NUMEXPR` does not evaluate to a number, a non-fatal run-time error occurrs.

Example.

```
p(X,Y) <- X is Y+1.
```

7.2.8 Relational Operators, $<, >, \geq$

Synopsis: `NUMEXPR1 relop NUMEXPR2`

Parameters:

`NUMEXPR1` term or numerical expression which must evaluate to a number during run-time.

`NUMEXPR2` term or numerical expression which must evaluate to a number during run-time.

`relop` a relational operator out of $<, >, \geq$.

Low Level Name: `sub, jmpz, jmpg, jmp`

Result:
The comparison succeeds, if the relational operation evaluates to TRUE.

Example.

```
p(X,Y) <- X < Y, Y >=5.
```

7.2.9 $isvar/1

Synopsis: `$isvar(T)`

Parameter:

`T` term

Low Level Name: `is_var`

Description. This built-in predicate succeeds, if `T` is a variable or bound to a variable.

Note. Global variables are initially bound to ordinary variables. Therefore, this test is also suited for global variables.

Example.

```
p(X) <- $isvar(X),...
```

Using `$isvar` allows to enter that clause only, if `X` is instantiated to a variable, e.g., if called from the (sub-) goal `<- p(Y).`

7.2.10 $isnonvar/1

Synopsis: `$isnonvar(T)`

Parameter:

`T` term

Low Level Name: `isnon_var`

Description. This built-in predicate succeeds, if `T` is *not* a variable or bound to a variable.

Example.

```
p(X) <- $isnonvar(X),...
p(X) <- $isvar(X),...
```

The built-in predicates `$isvar` and `$isnonvar` can be used to distinguish clauses which can be entered, if a variable is bound (here: first clause), or not (second clause).

7.2.11 $isconst/1

Synopsis: `$isconst(T)`

Parameter:

`T` term

Low Level Name: `is_const`

Description. This built-in predicate succeeds, if `T` is bound to a symbolic constant or a string.

Note. Numbers and complex terms (e.g., `f(a,b)` are no symbolic constants.

Example.

```
p(X) <- $isconst(X),...
p(X) <- $isvar(X),...
```

The first clause can be entered, if `X` is bound to a constant, the second only, if `X` is bound to a complex term, e.g., when called with `<- p(f(a)).`.

7.2.12 $iscompl/1

Synopsis: `$iscompl(T)`

Parameter:

`T` term

Low Level Name: `is_compl`

Description. This built-in predicate succeeds, if `T` is bound to a complex term $f(t_1, \ldots, t_n)$, or a list.

Example.

```
p(X) <- $isconst(X),...
p(X) <- $iscompl(X),...
```

The first clause can be entered, if `X` is bound to a constant, the second only, if `X` is bound to a complex term, e.g., when called with `<- p(f(a)).`.

7.2.13 $isnumber/1

Synopsis: `$isnumber(T)`

Parameter:

`T` term

Low Level Name: `is_number`

Description. This built-in predicate succeeds, if `T` is bound to a number.

Example.

```
p(X) <- $isnumber(X),...
```

7.3 Built-ins for Control

7.3.1 $eqpred/1

Synopsis: `$eqpred(N)`

Parameter:

`N` number of literal to check. `N` must be in the range between 1 and the length of the clause (including all built-in predicates).

Low Level Name: `eqpred/4`

Description. A built-in of this form fails, whenever the n-th literal in the current clause is preceeded by an identical one (syntactically equal) in the current tableau. This check is performed, when the built-in is executed (in contrast to SETHEO's regularity constraints).

Note. This built-in can be used to implement a weak form of regularity checks.

Example.

```
L1:- $eqpred(1),     /* eq pred check for the head literal L1.   */
     L3, $eqpred(3), /* eq pred check for the literal ~L3.       */
     L5, $eqpred(5), /* eq pred check for the literal ~L5.       */
               :
               :
```

7.3.2 $fail/0

Synopsis: `$fail`

Low Level Name: `fail`

Result:
This built-in always fails.

Description. This built-in unconditionally fails and thus represents an unsolvable subgoal.

Example.

```
p(X) <- q(X,Y),$write(Y),$write("\n"),$fail.
```

This clause tries to find all solutions for the subgoal `q` and prints the corresponding substitution of the variable `Y`.

7.3.3 $cut/0,$precut/0

Synopsis: `$precut, ..., $cut`

Low Level Name: `pre_cut, cut`

Description. These two built-ins implement the PROLOG-cut "!". The PROLOG clause

```
p(X) :- q(X),
        !,
        r(X).
```

is operationally equivalent to the LOP clause

```
p(X) :- $precut,
        q(X),
        $cut,
        r(X).
```

The `$precut` always must be the *first* tail-literal of a clause.

Note. The Prolog Clause

```
p(X) :- not q(X),
        r(X).
```

is operationally equivalent to the following sequence of LOP clauses

```
p(X) :- not_q(X),
        r(X).

not_q(X) :- $precut,q(X),
         $cut, $fail.
```

Example.

```
p(X) :- ...
p(X) :- $cut,$precut.
p(X) :- $write("cannot be entered\n").
```

7.3.4 $stop/0

Synopsis: `$stop`

Low Level Name: `stop`

Description. This built-in unconditionally stops the run of the SAM with a success message. Any illegal usage of this built-in destroys soundness of SETHEO.

Example.

```
p(X) <- $write("End of run"),$stop.
```

7.3.5 $monitor/0

Synopsis: `$monitor`

Low Level Name: `break`

Result:
This built-in always succeeds (if it returns at all).

Description. When this built-in is executed, the SAM immediatedly jumps into the interactive monitor mode (unless the SAM is called with `-batch`). In that mode, the user can continue `cont`, quit `quit`, or display a variety of (low-level) data. For details of all monito-commands see Section ??.

Example.

```
p :- q(X),$monitor,r(X).
```

7.4 Built-ins for Input-Output

7.4.1 $ptree/0

Synopsis: `$printtree`

Low Level Name: `printtree`

Result:
This built-in always succeeds.

Description. When this built-in is encountered, it appends the current tableau to the file *file*.tree.

Notes. (1) In the current version, **xptree(1)** can only display one proof at a time. (2) The entire tableau is printed, not just the part, starting with the current clause.

Example.

```
find_all_proofs :- q(X,Y),$ptree,$fail.
```

When called, this clause tries to find all solutions for the subgoal `q(X,Y)`, and appends the proof for each solution to the tree-file.

7.4.2 $tell/1

Synopsis: `$tell(T)`

Parameter:

`T` term or string

Low Level Name: `tell`

Result:

This built-in succeeds, if `T` is bound to a valid file-name, and a file with that name could be opened. Otherwise a fail occurs and a warning message is issued.

Description. When this built-in is encountered, the SAM tries to open a file with the given name in the "append mode" ("+a"). Subsequent outputs (e.g., by `$write` or `$dumplemma`) will be directed into that file. If a named file has been open previously, that file is closed prior to opening the new one.

In contrast to PROLOG, no nested `$tell`'s are allowed.

Note. Since strings and terms are treated likewise, `$tell(foo)` and `$tell("foo")` have the same effect.

Example.

```
p(X) :- $tell("output"),$write(X),$told.
```

7.4.3 $told/0

Synopsis: `$told`

Low Level Name: `told`

Result:

This built-in always succeeds

Description. When encountered, this built-in closes a named file which have previously been opened by `$tell(T)`.

Example.

```
p(X) :- $tell("output"),$write(X),$told,$write("done").
```

This clause writes the current values of `X` into the file `output`. Then, the word `done` is printed to stdout.

7.4.4 $write/1

Synopsis: `$write(T)`

Parameter:

`T` term to be printed

Low Level Name: `out`

Result:
This built-in always succeeds

Description. The printable representation of the given term is printed. Logical variables are printed in the form `X_`*number*, where *number* identifies the variable in a unique way. Please note that these numbers may differ from run to run.

Quoted special characters (e.g., `\n`) in strings are expanded.

Example.

```
p(X) :- $write("X="), $write(X), $write("\n\007").
```

When called with `<- p(f(a))` this clause will print `X=f(a)`, a new-line, and will ring the bell.

7.5 Built-ins for Bounds and Sizes

7.5.1 $get*bound*/1

This section describes a number of built-ins which allow to access the current (or maximal) values of the search bounds.

Synopsis: `$get`*bound*`(N)`

Parameter:

`N` number or variable

Low Level Name: `getdepth`, `getinf`, `getlocinf`, `get_maxinf`, `get_maxfvars`, `get_maxtc`, `get_maxsgs`

Description. A built-in of this group obtains the current or maximal value of a search-bound and unifies this value with the parameter `N`.

`$getdepth(N)` gets the maximum tableau depth (allowed for a final tableau) minus current depth.

`$getinf(N)` gets the current number of inferences (i.e. the number of inferences needed to derive the current tableau).

`$getlocinf(N)` gets the current value of the local inference bound.

`$getmaxinf(N)` obtains the maximum number of inferences (allowed for a final tableau). This value is set at the beginning of a search level (when iterative deepening is used), and then kept constant.

`$getmaxfvars(N)` gets the maximum number of free variables (allowed for a final tableau).

`$getmaxtc(N)` gets the maximum termcomplexity (allowed for a final tableau).

`$getmaxsgs(N)` gets the maximum number of open subgoals (allowed for a final tableau).

Example.

```
p(X) :- $getdepth(D), D > 5, q(X).
```

This clause can be used only, if the remaining allowable depth is greater than 5.

7.5.2 $set*bound*/1

This section describes a number of built-ins which allow to destructively set the current (or maximal) values of the search bounds. All assignments to the bounds are backtrackable, i.e., their effect is undone as soon backtracking occurs over the corresponding built-in.

Synopsis: `$set`*bound*`(N)`

Parameter:

`N` term which must evaluate to a non-negative number.

Low Level Name: `set_depth, set_inf, set_maxinf, set_locinf, set_maxfvars, set_maxtc, set_maxsgs`

Result:
These built-ins always succeed.

Description. A built-in of this group sets the current or maximal value of a search-bound.

`$setdepth(N)` sets the bound to the maximum tableau depth (allowed for a final tableau) minus current depth.

`$setinf(N)` sets the number of inferences.

`$setlocinf(N)` sets the number of local inferences.

`$setmaxinf(N)` sets the maximum number of inferences (allowed for a final tableau).

`$setmaxfvars(N)` sets the maximum number of free variables (allowed for a final tableau).

`$setmaxtc(N)` sets the maximum termcomplexity (allowed for a final tableau).

`$setmaxsgs(N)` sets the maximum number of open subgoals (allowed for a final tableau).

Side-effects. These built-in effect global registers of the SETHEO Abstract Machine SAM. Therefore, a wrong usage of these built-ins can have undesired effects and can destroy the completeness of the proof procedure.

Note. These built-ins are used to adapt the iterative deepening, or to implement Meta-interpreters. For details and examples refer to Section **??**.

Example.

```
p(X) :- $getdepth(D),D1 is D+1,$setdepth(D1),q(X),$setdepth(D).
```

When this clause is called, then its subgoal `q` can use the same resources (here, the depth), as the clause itsself.

7.5.3 $size/2

Synopsis: `$size(T,N)`

Parameters:

`T` term

`N` number or variable

Low Level Name: `size`

Description. This predicate calculates the size of the term `T` and unifies the result with the second parameter `N`. The size of a term is the number of its constants, variables, and function symbols.

Example.

```
p <- $size(f(a,X),2)
p <- $size(f(a,g(Y)),2)  /* will fail */
```

7.5.4 $tdepth/2

Synopsis: `$tdepth(T,N)`

Parameters:

`T` term

`N` number or variable

Low Level Name: `tdepth`

Description. This predicate calculates the depth of the term `T` and unifies the result with the second parameter `N`. The depth of a term is maximal length of its paths.

Example.

```
p <- $tdepth(f(a,X),2).
p <- $tdepth(f(a,g(Y)),3).
```

7.6 Built-ins for Lemmata

7.6.1 $genlemma/2

Synopsis: `$genlemma(N1,N2)`

Parameters:

`N1` If this number is $\neq 0$, the generation of a unit-lemma is considered. Otherwise, this predicate just succeeds.

`N2` If a unit-lemma is stored in the index, this number will be stored together with the lemma. This information can then be used when unit-lemmata are being used (e.g., by **uselemma/1**).

Low Level Name: `genlemma`

Result:
This built-in predicate always succeeds. Memory overflow and wronly instantiated parameters will result in a non-fatal run-time error.

Description. This built-in generates a unit-lemma `H<-.` for the head `H` of the current clause, if all of the following conditions hold:

- both arguments are instantitated to a number,
- the value of the first argument is $\neq 0$,
- The solution of the head of the current subgoal does not involve any reduction steps above the current node in the tableau. (This means, that we really have a unit-lemma).
- The current head (with its current instantiation) is not subsumed by any unit-lemma in the lemma store.

The lemma will be annotated by the value of the second argument (must be a number).

If the unit-lemma subsumes one or more unit-lemmata in the lemma-store, these unit-lemmata are deleted.

Side-effects. If the **sam(1)** is called with the inline-command flag `-lemmatree`, the proof tree for the current lemma (if it gets stored in the index) is appended to the proof-tree file.

Note. The **genlemma/1** built-in must be the last subgoal of a given clause or contrapositive. Otherwise, correctness of SETHEO is not ensured, if the lemmata are used.

Example 1.

```
q(X,Y) :- p(X),r(X,a),$genlemma(1,5).
```

In this example, a unitlemma `q(X,Y)<-.` — with the current substitutions of `X` and `Y` — is generated and labeled with the number 5.

Example 2.

```
q(X,Y) <- p(X),r(X,a),$getdepth(D),$genlemma(1,D).
```

In this example, the generated unit-lemma is labeled with the current depth. When lemmata are selected for usage (`$uselemma/1`), one can select only those unit-lemmata which have been generated on a lower depth.

7.6.2 $genulemma/3

Synopsis: `$genulemma(N1,N2,N3)`

Parameters:

`N1` This number indicates the number of the literal for which a unit-lemma is to be generated. `N1` must be in the range between 1 and the length of the clause (including built-in literals).

`N2` If this number is $\neq 0$, the generation of a unit-lemma is considered. Otherwise, this predicate just succeeds.

`N3` If a unit-lemma is stored in the index, this number will be stored together with the lemma. This information can the be used when unit-lemmata are being used (e.g., by **uselemma/1**).

Low Level Name: `genulemma`

Result:

This built-in predicate always succeeds. Memory overflow and wronly instantiated parameters will result in a non-fatal run-time error.

Description. This built-in generates a unit-lemma `L<-.` for a literal `L` in the current clause. Its number is given as the first argument. The lemma is generated, if all of the following conditions hold. The lemma will be annotated by the value of the third argument (must be a number).

- all arguments must be instantitated to numbers, otherwise, a run-time error is generated.
- the first argument must be instantiated to a number in the range between 1 and the length of the clause (including built-in literals).
- the value of the second argument is $\neq 0$,
- the current instantiation of the given literal is not subsumed by any unit-lemma in the lemma store.

If the unit-lemma subsumes one or more unit-lemmata in the lemma-store, these unit-lemmata are deleted.

Notes.

- the range of the first argument is NOT checked
- it is not checked, if the solution of the given literal required reduction steps above the current node in the tableau. Therefore, it is easily possible to generate unsound proofs, if yet unsolved literals are kept as lemmata.
- all built-in literals (including this one) count as literals

Example.

```
q(X,Y) :- p(X),$genulemma(2,1,5),r(X,Y).
```

generates a lemma `p(a)<-.`, if X is instantiated to `a`. The lemma carries the number 5.

7.6.3 $uselemma/1

Synopsis: `$uselemma(N)`

Parameter:

`N` number

Low Level Name: `uselemma`

Description. during run-time, only lemmata from the index may be used which carry a number which is smaller than the given parameter.

This built-in always succeeds. If the pushed lemmata and the remaining alternatives are to be tried, this built-in must be directly followed by a **fail**. This built-in tries to use lemmata from the lemma-store. This built-in must be used within a clause of a specific structure as shown below. When called, this built-in extracts all lemmata from the index and marks them as possible choices, which fulfill the following conditions:

1. the lemma in the index must not be marked "deleted".
2. the lemma must be unifiable with the head of the current clause.
3. the number stored with the lemma must be smaller (<) than the given parameter.

After adding the choices, this built-in succeeds, If the selected lemmata are to be tried, a fail must be executed after this built-in. Then, the lemmata are tried and then all other or-branches for the current subgoal which have not yet been touched are tried.

Example

The following example illustrates the usage of this built-in.

```
...
p(a,b) <-..
p(b,c) <-..

p(_,_) :- $uselemma(5),$fail.

p(e,f) <-.
..
```

If the clauses are not reordered, SETHEO first tries the alternatives `p(a,b)..` and `p(b,c)...` Then it extracts all suitable lemmata from the lemma store.

These lemmata are pushed upon the stack as additional alternatives which are tried (caused by the fail), before `p(e,f)..` is tried.

7.6.4 $dumplemma/0

Synopsis: `$dumplemma`

Low Level Name: `dumplemma`

Result:
This built-in always succeeds.

Description. This built-in dumps all lemmata in the lemma-store onto the current output-file (as set by **tell/1**), or on stdout. Lemmata are printed in a LOP-like syntax. Therefore, the output can directly be processed by **inwasm(1)**. Lemmata which have

been marked deleted are preceeded by a #. This option is present for debugging and testing purposes and may be removed in future versions.

Example.

```
printlemma :- $tell("out"),$write("Lemmata:\n"),
              $dumplemma,$told.
```

All lemmata present in the index are printed into the file `out`.

7.6.5 $getnrlemmata/1

Synopsis: `$getnrlemmata(N)`

Parameter:

`N` this parameter unifies with the number of lemmata currently in the lemma index.

Low Level Name: `getnrlemmata`

Result:
This built-in fails, if the unification fails.

Description. This built-in unifies the given argument with the number of lemmata which are currently stored (and not deleted) in the lemma store.

Example.

```
p(X) :- $getnrlemmata(Y), X > Y,...
```

7.6.6 $delrange/2

Synopsis: `$delrange(N1,N2)`

Parameters:

`N1` lower bound of the range to delete

`N2` upper bound of the range to delete

Low Level Name: `delrange`

Result:
This built-in succeeds, if both parameters are instantiated to numbers.

Description. This built-in deletes all lemmata in the lemma store with annotated values between the first and the second parameter (inclusive). Both parameters must

be instantiated to a number.

Example.

```
p :- delrange(3,5).
```

7.7 Built-ins for controlling Counters

7.7.1 $initcounters/1

Synopsis: `$initcounters(N)`

Parameters:

`N` number of counters to provide

Low Level Name: `init_counters`

Description. This built–in provides `number` memory cells for storing counter values which are not deleted in case of backtracking.

Notes.

1. The memory cells are not initialized. This ensures that they are not reinitialized resp. their values are not deleted in case of backtracking.
2. Make sure `$initcounters` is the first subgoal of the query. This will prevent overwriting the stored values in case of backtracking.
3. There should be only one subgoal `$initcounters` in a query. After the second call of `$initcounters` the first counter block is not accessible any more.

Example.

```
:- $initcounters(5), p(X), q(X).
```

7.7.2 $setcounter/2

Synopsis: `$setcounter(N,T)`

Parameters:

`N` number of counter to access

`T` value to store in the `N`–th counter

Low Level Name: `set_counter`

Result:

If `N` is not a number or less than 1 or greater than the number of provided counters this built–in returns an error state. If `T` is not a number this built–in returns an error state as well. Otherwise this built–in succeeds.

Description. `$Setcounter` first checks if `N` is a correct index to the block of provided counters and if `T` is a number. If so, `$setcounter` stores the value of `T` in the `N`–th counter.

Notes. The values to be stored have to be numbers and they have to be in the range $[-32768, 32767]$.

Example.

```
:- $initcounters(5), p(X), $setcounter(2,10), q(X).
```

7.7.3 $getcounter/2

Synopsis: `$getcounter(N,T)`

Parameters:

`N` number of counter to access

`T` term to unify the value of the `N`–th counter with

Low Level Name: `get_counter`

Result:
If `N` is not a number or less than 1 or greater than the number of provided counters this built–in returns an error state. If `T` is not unifiable with the value of the `N`–th counter this built–in fails. Otherwise `$getcounter(N,T)` succeeds.

Description. `$Getcounter` first checks if `N` is a correct index to the block of provided counters. If so, `$getcounter` tries to unify `T` with the value of the `N`–th counter.

Notes.

1. Since `$initcounters` does not initialize the memory cells `$getcounter(N,T)` should only be called after `$setcounter` has been applied to the `N`–th counter.
2. After successful unification `T` is interpreted as a number.

Example.

```
:- $initcounters(5), p(X), $setcounter(2,10), q(X), $getcounter(2,T).
```

7.8 Built-ins for Processing Terms

7.8.1 $functor/3

Synopsis: `$functor(T,F,N)`

Parameters:

`T` term

`F` functor

`N` arity

Low Level Name: `functor`

Description. see PROLOG.

`johann`

Example.

`XXX`

7.8.2 $arg/3

Synopsis: `$arg(T,N,A)`

Parameters:

`T`

`N`

`A`

Low Level Name: `arg`

Result:

Description. see PROLOG

`johann`

Example.

`xxx`

8. How To ...

8.0.3 Reporting SETHEO Errors

If the SETHEO-system behaves strangely (e.g. core dump, irregular output, funny error messages) and it is suspected that one or more modules of the SETHEO system contain a bug, the user is adviced to report this bug to the SETHEO's development group in Munich. Before doing so, the user should carry out the following steps.

1. Check, if an error message is produced by one (or some) of the modules. If an error is reported by one of the modules, one should not run the others with faulty output. E.g. if the **inwasm** reported an error, the assembler **wasm** and the SAM should not be executed, even if code has been produced.
2. Check, if the formula file is non-empty.
3. Check, if at least one start clause (with a `<-` or a `?-` in the beginning) is present (and not being removed by **inwasm**'s purity reduction).
4. Check the command-line parameters used on this example. Does the error occur with a different parameter setting as well?
5. Check, if the error is reproducable.
6. If logic-programming features of LOP and built-ins are used: is the subgoal-reordering (**inwasm** `-sgreord`) and the clause reordering (**inwasm** `-clreord`) turned off where necessary? Do all built-in predicates start with a `$`? Be aware of the fact that all clauses with a `<-` separator are being fanned (if there is at least one Non-Horn clause in the formula) and their subgoals are reordered. When subgoals are reordered, built-in predicates are placed at the beginning of the clause.
7. If logic-programming features of LOP and built-ins are used: are the required variable bindings for the built-in predicates OK ? E.g. a `X is Y + 1` requires `Y` to be instanitated to a numeric value.
8. Try to minimize the example by leaving out clauses or literals or by simplifying them, until the error disappears. The smaller the resulting formula, the easier it is to detect a possible error (both for you and the SETHEO development group).

If all of the above steps have been tried to no avail, send in your error report to

`setheo@informatik.tu-muenchen.de`

The report should contain:

1. name and address of the user,

2. version number of the SETHEO system (any modifications of the code by the user?),
3. hardware platform and version of the operating system,
4. a short description of the error,
5. a complete set of parameters which have been used to produce the error,
6. a run-time protocol (e.g. the `.log` file) and
7. the formula which produced the error (as small as possible).

The SETHEO development group in Munich then will have a look at the problem and will try to solve it ASAP.

8.1 Set Parameters

In this section some hints are given how to set SETHEO's parameters in case of special problems. But these are just rules of thumb, sometimes you might find better parameter settings by yourself. For a detailed parameter description see Section 5.4.

Logic Programming: In the compilation phase the *subgoal reordering*, the *clause reordering* and the generation of *overlap constraints* should be switched off. So call the **inwasm** with the **–nosgreord –noclreord** options and make sure not to call **–oconsx** (**–ocons** is possible). .

Small Formulae: Small formulae do not cause any problems. Call the **inwasm** with the **–cons** option, the **wasm** with the **-opt** option and the **sam** with the **–dr –cons** options. These are good standard parameters.

Large Formulae: When using large formulae the preprocessing is very time consuming. The preprocessing can be fastened if *overlap constraints*, *subsumption constraints* and *link subsumption* are avoided. If there are only a few rules but many facts, the **inwasm** can be called with the **–rsubs [n] –rlinksubs [m]** options because this options do not consider facts. Otherwise the options **–[r]subs [n]** and **–[r]linksubs [m]** should be avoided. The option **–overlap[x] [n]** should generally not be used for proving large formulae.

Propositional Logic:

Long Proofs: During long proofs it is important to reduce the search space. There are several bounds to reduce the search space[1]. In general you do not know the size of the proof, so an iterative bound might be best.
First try the combination of *iterative deepening* and *constraints*: Call the **inwasm** with the **–cons** option, the **wasm** with the **–opt** option and the **sam** with the **–dr –cons** options. Iterative deepening is in most cases the best iterative bound, and constraints additionally reduce the search space.
If you can estimate the number of inferences, the number of free variables, the termcomplexity or the number of open subgoals you can use this knowledge by

[1] See Section 5.4.4.

setting the corresponding bounds. Note that the free variables bound is not complete in combination with antilemmata.
There are two more iterative bounds: The *iterative inferencing* and the *iterative local inferencing*. So you can also call the SAM with the **–ir** resp. the **–locir** option. Note that only one iterative bound is possible at the same time. For using the *iterative inferencing* it is very useful to estimate the depth of the proof and call the SAM with **–ir 1 –d n –cons** where n is the estimated depth.
During large proofs the **sam** might stop with an *overflow error*. In such a case you have to start the SAM again with the overflowed memory area enlarged. Read Section 5.4.4 for the enlarging options.
The most frequent overflow error is the *constraint stack overflow* error. In this case you can either reduce the number of generated antilemma constraints[2]. The option **-anl [n]** reduces the generated antilemmata to such for which at least n inference steps have been made. The default is n=25, so if you want to reduce the number of generated antilemma constraints set n>25. This should decrease the runtime, but will increase the number of inferences.
In some cases the *foldup* mechanism leads to good results. This mechanism generates lemmata from closed branches. Call the **inwasm** with the **–cons –foldup** options and the **wasm** and the **sam** as above. But note that all methods to reduce the search space will reduce the number of generated lemmata, too. So don't be surprised if the combination of such two powerful methods as foldup and antilemmata doesn't lead to the expected results. In a few cases the combination works even worse than antilemmata or foldup at its own. When using foldup it might eventually be better to call the **sam** with the **–noanl** option, just try.

8.2 Selection of Start Clauses

Per default, SETHEO takes all clauses which contain negative literals only, as possible start clauses. In LOP syntax, such clauses have the following form:

$$\texttt{<-}L_1, \ldots, L_n.$$

with atoms (non-negated literals) L_i. These clauses are also referred to as *queries*.

The Model Elimination Calculus, however, allows to select an *arbitrary* clause as start clause. This can be of interest if the conjecture to be proven is of the form $A_1 \wedge \ldots \wedge A_n$. If a true goal-oriented search is intented, the search should start with this clause. On the other hand, axioms like `<-equal(X,succ(X)).` should in general not be used as a starting clause.

Cases like this can be handled by SETHEO. In the following, we describe a way to (a) disable a query, and to (b) convert an arbitrary clause into a query.

[2]There is no possibility to reduce the other kinds of constraints, but its usually the antilemma constraints that cause a constraint stack overflow.

8.2.1 Disabling a Query

A query (as shown above) can be disabled by replacing that clause with the following one (for any $1 \leq i \leq n$):

$$\sim L_i \texttt{<-} L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n.$$

This rule cannot be used as a starting clause by SETHEO. Nevertheless, all necessary contrapositives are being generated.

Please note that disabling starting clauses may lead to incompleteness.

8.2.2 Converting a Clause into a Query

Given a clause

$$H_1; \ldots; H_m \texttt{<-} L_1, \ldots, L_n.$$

with atoms (non-negated literals) L_i, H_i. Then this clause can be converted into a query (starting clause) by replacing this clause by:

$$\texttt{<-} \sim H_1, \ldots, \sim H_m, L_1, \ldots, L_n.$$

For this clause, all contrapositives are generated.

8.3 Assembly and Display of Answer Substitutions

In contrast to PROLOG systems, SETHEO does not automatically display answer substitutions of the variables in the query of a logic program or a theorem. Answer substitutions, however, are always present in the proof, SETHEO generates. They can be displayed, using the **xptree** tool.

A more convenient and elegant way to display the answer substitutions (both for the Horn and Non-Horn case) can be easily accomplished, using SETHEO's logic programming facilities.

In the following, we will describe, how it is possible to print one answer substitution or all substitutions in the Horn and Non-Horn case.

8.3.1 The Horn Case

For a formula (or logic program) which only consists of Horn-clauses, the answer substitution consists of exactly one substitution for each variable in the query. E.g., for a query `<- p(X,Y)`, an answer substitution could be $X \backslash a, Y \backslash b$.

Such a query can easily be instrumented in such a way that such answer substitutions are printed, as soon as a proof could be found. We just print the substitution, using the *$write/1* built-in. In our above example, one would have to write:

```
?- p(X,Y),$write("X="),$write(X),$write(" Y="),$write(Y),$write("\n").
```

Please note that the `<-` now has been converted into a `?-` in that clause in order to prevent reordering of subgoals (i.e., to prevent the output of the variables before the main goal is called).

If you want to print all answer substitutions whcih can be found within given bounds, you have to add a `$fail` at the end of the clause. This forces SETHEO to backtrack, after a proof has been found.

8.3.2 The Non-Horn Case

For a non-horn formula, the answer substitution can be *disjunctive*. As an example, consider the following (non-Horn formula):

```
<- p(X).
p(a); p(b)<-.
```

The disjunctive answer substitution in that case is

$$X = a \vee X = b.$$

If we want to print such an answer substitution, our query must be transformed as follows:

```
?- $ANS := [], query, $write("X= "), $write($ANS), $Write("\n").  /*(1)*/

query :- p(X), $ANS := [X|$ANS].
~p(X) :- $ANS := [X|$ANS].
```

The rest of the formula remains unchanged. When SETHEO is started and finds a proof, it prints the list of all disjuncts of variable substitution. In our case, we get: `X= [a|[b|[]]].`

If a more convenient output is needed, however, additional clauses must be added in order to convert the list into a correct form. Note, that additional built-ins are needed to bypass the bounds, and to keep the number of inferences in the proof correct. The following program produces the desired output:

```
?- $ANS := [],
      query,
      $getinf(I),
      print_answersubst($ANS),
      I1 is I -1,
      $setinf(I1).  /*(1)*/

query :- p(X), $ANS := [X|$ANS].
~p(X) :- $ANS := [X|$ANS].

p(a);p(b)<-.

print_answersubst([X]) :- $write("X="),$write(X),$write("\n").
```

```
print_answersubst([X|Y]) :-
        $setdepth(1),$setinf(1),
        $write("X="), $write(X), $write(" ; "),
        print_answersubst(Y).
```

If more than one query exists in the formula, this piece of code can easily be generalized.

8.4 Iterative Deepening

8.5 Generating and Using Unit-lemmata

8.6 Defining New Built-Ins

9. What If ...

This Chapter contains hints of what to do if something goes wrong, e.g.

- what to do in cases of errors during compilation/assembly?
- setheo cannot find a proof
- iterative deeping reaches large bounds too quickly
- core-dump of a program
- handling large formulae
- SAM error messages

This Chapter also contains a table with limitations and default values of the SETHEO system, and a form to report a SETHEO bug.

9.1 SAM Error Messages

There are different types of errors which might occur during the SAM's runtime. The minor ones only cause a message printed on the screen. The heavy errors additionally lead to a system interrupt. If necessary, the hexadecimal code of the trouble causing object is displayed, too. The problems can be divided into three classes:

1. user dependend problems
2. internal problems
3. implementation problems

User dependend problems are either caused by the user or can easily be avoided by the user. *Wrong parameter choices* or *overflow errors* belong to this class. Examples for **wrong parameter choices** to the SAM are:

- The name of the code file together with extension **.hex** does not exist. *What you can do:* Check if you spelled the name correctly and if you left out the extension.
- The name of the code file is missing. *What you can do:* Add the code file.
- An option was requested which is not available in your current version of SETHEO. *What you can do:* If you need this option, this becomes a problem for a programmer because you have to change some compiler flags and compile the SAM again. If possible, just don't request this option.
- You have made a typing mistake concerning options or parameters to options. *What you can do:* Check the spelling of what you typed and maybe look at Section 5.4.4 again.

Some of the built–ins also require special kinds of parameters, e.g. the arithmetic built–ins expect number values. *What you can do:* You have to repair this kind of wrong parameter choices within the **.lop**–file. The error message tells you what to change. Don't forget to call the **inwasm** and the **wasm** again before calling the SAM.

Overflow errors are not caused by the user but they are avoidable by the user. An overflow error occurs if one of the SAM's memory areas is too small. Overflow errors always cause the SAM to interrupt. You can still get statistical information if you are interested in, but there is no way to go on proving. *What you can do:* Quit the program and start it again with the overflowed memory area enlarged. Eventually read Section 5.4.4 how to enlarge memory areas.

Internal problems are memory allocation problems or limit exceeding problems. **Memory allocation problems** can be the following:

- The operating system does not provide enough space to allocate the SAM's memory areas or the address of one of the memory areas is out of range. *What you can do:* Reduce the size of the memory areas. Use the same options as for enlarging the memory areas (see Section 5.4.4).
- The operating system does not provide enough space to allocate memory for storing antilemmata or the address of the allocated memory is out of range. *What you can do:* Reduce the number of generated antilemmata. Call the **-anl [number]** option with *number* enlarged. The default value for *number* is 25 (see Section 5.4.4).
- The operating system does not provide enough space to open an output file. *What you can do:* Reduce the size occupied by the SAM, i.e. reduce the size of the memory areas (see above).

The following problems are **limit exceeding problems**:

- A number value gets bigger than 32767 or less than –32768 and is truncated to a short value. (Remember the SAM works only with short arithmetic). *What you can do:* Try to avoid refering to this value.
- A choicepoint contains too many alternatives for the random reordering instruction. *What you can do:* If you want to repair this problem you have to change a constant within the SAM's source code, so this becomes an implementation problem.

Implementation problems arise from wrong or incomplete changes to the SETHEO system. Messages about illegal tags, illegal instructions or unavailable bounds belong to this kind. *What you can do:* Errors concerning illegal tags are hard to debug. So think very carefully what you are changing. Errors concerning illegal instructions might result from spelling mistakes or from not having made the corresponding changes in **inwasm** and **wasm.** Errors concerning unavailable bounds should be detectable using a usual debugging tool.

9.2 Return values of the SETHEO system

0 print usage or print verbosity levels `inwasm -verbose`

0 no input file specified

1 error in command-line parameters

2 cannot open input or output files

3 command parameters are not consistent

1 syntax errors occurred (abort after parsing)

4 purity removed all clauses. "Fail" is generated.

1 errors occurred during preprocessing (before code generation)

0 successful termination

7 code generation: out of memory

7 code generation: undefined labels during pass2

11 code generation: symbol table overflow

9 code generation: optimizer internal: must be a label

1 usage

1 no input file specified

1 error in command-line arguments

2 could not open input- or output files

1 syntax error during pass1

1 syntax error during pass2

0 successful termination

1 illegal nesting of include's

1? out of memory

3 error during loading hex-file

1 usage

9 illegal parameter

5 input file missing

1 error in command-line parameter

2 no input file

1 cannot open input file

0 quit from interactive mode

5 CPU-time or real-time failure

? exit value of "sam_error()" in case, no singals are enabled.

0 SUCCESS (instr_result)

1 FAILURE (instr_result)

2 ABORT (instr_result)

3 error during SAM-execution (instr_result)

4 memory overflow

5 total failure

6 failure (depth bound reached)

7 failure (copy bound reached ???)

8 failure (inference bound reached ???)

9 failure (local inference bound reached ???)

10 failure (constraints)

10. Installing the Sources

10.1 Introduction

This Section is about how to install the SETHEO sources at your own computer. For installing the binaries refer to Chapter 3.

10.2 Getting the Sources

10.3 Installing the Sources

10.4 Migration to New Platforms

Bibliography

[Goller *et al.*, 1994] Chr. Goller, R. Letz, K. Mayr, and J. Schumann. SETHEO V3.2: Recent Developments (System Abstract) . In *Proc. CADE 12*, pages 778–782, June 1994.

[Harrison, 1996] J. Harrison. Optimizing proof search in model elimination. *CADE-13*, LNAI 1104, p. 313–327, Springer, 1996.

[Ibens and Letz, 1996] O. Ibens, R. Letz. Subgoal Alternation in Model Elimination. Technical Report, Institut für Informatik, Technische Universität München, 1996, submitted to TABLEAUX'97.

[Letz *et al.*, 1992] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[Letz, 1993] R. Letz. *First-Order Calculi and Proof Procedures for Automated Deduction.* Dissertation, Technische Hochschule Darmstadt, 1993.

[Letz *et al.*, 1994] R. Letz, K. Mayr, and C. Goller. Controlled Integration of the Cut Rule into Connection T ableau Calculi. *Journal Automated Reasoning (JAR)*, (13):297–337, 1994.

[Letz and Ibens, 1996] R. Letz, O. Ibens. Properties of Iterative Deepening Procedures. Technical Report, Institut für Informatik, Technische Universität München, 1996, submitted to CADE-14.

[Loveland, 1978] D. W. Loveland. *Automated theorem proving: A logical basis.* North Holland, New York, 1978.

[Schumann and Letz, 1990] J. Schumann and R. Letz. PARTHEO: a High Performance Parallel Theorem Prover. In M. E. Stickel, editor, *CADE10*, Lecture Notes in Artificial Intelligence, pages 40 – 56. Springer, 1990.

> PARTHEO, a sound and complete or-parallel theorem prover for first order logic is presented. The proof calculus is model elimination. PARTHEO consists of a uniform network of sequential theorem provers communicating via message passing. Each sequential prover is implemented as an extension of Warren's abstract machine. PARTHEO is written in parallel C and running on a network of 16 transputers. The paper comprises the system architecture, the theoretical background, details of the implementation, and results of performance measurements.

[Schumann *et al.*, 1990] J. Schumann, N. Trapp, and M. van der Koelen. SETHEO/PARTHEO: User's Manual. Technical Report TUM-I 9010. SFB342/7/90 A, Technische Universität München, SFB 342, 1990.

[Schumann, 1991] J. Schumann. *Efficient Theorem Provers based on an Abstract Machine*. PhD thesis, Technische Universität München, 1991.

[Schumann, 1994] J. Schumann. DELTA — A Bottom-up Preprocessor for Top-Down Theorem Provers, System Abstract. In *CADE 12*, 1994.

[Schumann, 1995] J. Schumann. Using SETHEO for Verifying the Development of a Communication Pro tocol in FOCUS – A Case Study –. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Proc. of Workshop Analytic Tableaux and Related Methods, Koblenz*, volume 918 of *LNAI*, pages 338–352. Springer, 1995.

This paper describes experiments with the automated theorem prover SETHEO. The prover is applied to proof tasks which arise during formal design and specification in FOCUS.

These proof tasks originate from the formal development of a communication protocol (Stenning protocol). Its development and verification in FOCUS is described in "C. Dendorfer, R. Weber: *Development and Implementation of a Communication Protocol – An Exercise in* FOCUS" [DW92]. A number of propositions of that paper deal with safety and liveness properties of the Stenning protocol on the level of traces. All given propositions and lemmata could be proven automatically using the theorem prover SETHEO.

This paper gives a short introduction into the proof tasks as provided in [DW92]. All steps which were necessary to apply SETHEO to the given proof tasks (transformation of syntax, axiomatization) will be described in detail. The surprisingly good results obtained by SETHEO will be presented, and advantages and problems using an automated theorem prover for simple, but frequently occurring proof tasks during a formal development in FOCUS, as well as possibly ways for improvements for using SETHEO as a "back-end" for FOCUS will be discussed.

[Stickel, 1988] M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[Sutcliffe *et al.*, 1994] G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In *Proceedings of the 12. International Conference on Automated Deduction (CADE)*, pages 252–266. Springer LNAI 814, 1994.

11. Glossary

SAM SETHEO Abstract Machine

fold-up

depth Depth in this context always means the A-literal depth of a Model Elimination Tableau.

12. Index