The Saturate System

Harald Ganzinger
Robert Nieuwenhuis
Pilar Nivela

December 4, 2002

Abstract

The Saturate system is an experimental theorem prover for first-order logic, primarily based on saturation. Saturate uses techniques of ordered chaining for arbitrary transitive relations, including orderings, equivalence relations and congruences, and integrates CNF transformation lazily into saturation.

Contents

1 Installation

Saturate runs under Quintus 3.2 (SunOS4.1, 5.3 and higher), under SICStus 2.1 (patch levels #9 or higher; SunOS 4.1 only)[1] and SICStus 3.x (3.0: patch levels #3 or higher; SunOS4.1, 5.3 and higher).

It also runs under SICStus 3.x (Linux and Windows).

1.1 Copying the System

You may obtain the system by ftp from ftp.mpi-sb.mpg.de as compressed tar file `pub/SATURATE/saturate.tar.gz` or, in a web browser, by clicking this link.

Then untar `saturate.tar` in some directory <saturate_install_dir> which generates a subdirectory `saturate` containing the system (in subdirectories called `papers`, `sources`, `doc`, and `examples`). The documentation can be found in the subdirectory `doc` (manual in hypertext format (subdirectory `Saturate`, file `Saturate.html`) or as a gzipped Postscript file `Saturate.ps.gz`).

1.2 Installing the System

Take the following steps

- cd <saturate_install_dir>/saturate/sources
- Replace in the file `sat.pl` the lines

  ```
  assert(saturate_directory( '/HG/local/provers/saturate' ) .
  davis_putnam( yes ).
  ```

  by

  ```
  saturate_directory( '<saturate_install_dir>/saturate' ) .
  davis_putnam( <yes/no> ).
  ```

  You may also want to adjust the configuration of the web browser to be used for displaying the help files. With netscape (Unix) or explorer (Windows) it should just work fine as is. Even if those settings are not correct that's not a problem, the Saturate system will function in any case.

  The flag for `davis_putnam` indicates as to whether the optional incremental Davis-Putnam-like satisfiability checker should be included in the system. The flag has no effect for Quintus- and SICStus 2.1-Prolog. The module only works under SICStus 3.x.

[1] This used to be the case up to version 2.4, but has not been verified for version 2.5.

For SICStus 3.x users it is possible that on your computer system "yes" will produce errors due to incompatibilities on the C-compiler and C-libraries level. In that case change the default to "no" and run the generation again. Saturate will function without that module, too, but will perform much worse in medium to large examples of near-propositional nature and in certain knowledge representation applications. If you want that module, you may obtain it via ftp. You will then have to re-compile them on your system, **using the same C-compiler that was used for installing SICStus 3.0**. Then move the resulting modules into directory <saturate_install_dir>/saturate/sources/CLPPB and say `davis_putnam( yes ).` before generating Saturate again.

- **For SICStus Prolog users only:** You need to have patch level #3 or higher for SICStus 3.0 and patch level #8 or higher for SICStus 2.1 (patch level #9 recommended). For later versions of SICStus 3.x any patch level should be fine.

  - On SPARC architectures include

    ```
    :- prolog_flag(compiling,_,fastcode).
    ```

    into your `.sicstusrc` file (or execute that query on top level) before installing the system if you want native code generation (the preferred mode).

  - Include

    ```
    setenv SP_COMPRESS_SAVE 1
    ```

    into your `.cshrc` file to save space when saving Saturate.

- cd to the `sources` subdirectory.

- Call Quintus Prolog or SICStus Prolog.

- Execute

  ```
  | ?- compile(sat), genSystem.
  ```

  (don't forget the dot at the end). Then, on demand, enter the name of a file to save the Saturate system on. Use that name on shell level to call Saturate lateron, or when restoring that state.

- Exit the Prolog system or start using Saturate right away.

- Don't remove the source files after system generation. Some of them will be needed and linked dynamically at runtime.

- If you would like to check as to whether the system operates correctly, execute

```
| ?- cd('<saturate_install_dir>/saturate/examples'),
     hgstuff:testsuite.
```

  As of November 2001 this runs for about 180 sec under SICStus 3.x on a 600Mhz Pentium IIII. The resulting behaviour should look like the one that is documented in the log-file <saturate_install_dir>/saturate/testsuite.log*.

2 Overview

The Saturate system is an experimental automated theorem prover for first-order logic primarily based on saturation up to redundancy. The theoretical basis of Saturate are rewrite techniques for transitive relations as described in (Bachmair & Ganzinger 1994*c*) and (Bachmair & Ganzinger 1995*b*). Clausal normal form transformation is integrated into Saturate in an attempt to keep the logical structure, in particular quantifiers and equivalences, visible. The system is written in Prolog and runs under Quintus and SICStus.

Saturate is a cooperation project between UPC Barcelona and MPI Informatik within the ESPRIT Basic Research Working Groups CCL and CCL II.

The main focus of the system is on

- the efficient treatment of transitive relations (Section 2.2.4) by term rewrite techniques (Section 2.2.1), and on
- the global restriction of the search space in saturation-based theorem proving by applying techniques for detecting redundancy (Section 2.3) of clauses and inferences.

Moreover, Saturate

- integrates a set of inference rules that are derived from the sequent calculus (Section 2.2.2) into saturation so as to not prematurely eliminate quaintifiers and other logical structure;
- provides methods for translating modal logics (Section 2.2.3) into first-order logic based on various semantic embeddings into Kripke structures, and
- supports AC-operators (Section 2.4) by providing AC-unification, AC-matching, AC-extensions of equalities, and AC-compatible orderings as a basis for a future integration of specific algebraic methods for theories such as abelian groups, rings and fields.

There are three basic kinds of problems for which Saturate can be used:

- for refutational theorem proving (Section 2.1.1)
- for the finite saturation of a consistent theory (Section 2.1.2), and
- for finding complexity bounds (Section 2.1.3) for decidable theories.

2.1 Applications

2.1.1 Refutational Theorem Proving

In refutational theorem proving we prove F by deriving a contradiction from $\neg F$. For simple cases, calling `prove` (Section 6.11) will do the job. The formula $\neg F$ will be saturated up to redundancy by a refutationally complete inference system (Section 2.2) until a contradiction is found. Alternatively, if $\neg F$ can be finitely saturated without producing a contradiction, F is not provable. Applying simplification and techniques for detecting redundancy, Saturate attempts to gain better control of its forward computation mode. The forward inferences (resolution, chaining, superposition) are restricted by ordering constraints and selection functions such that only maximal or selected atoms or terms need to be resolved or chained.

When one uses Saturate for refutational theorem proving, only a modest amount of simplification should be applied. If Saturate is at all able to find a proof, if will often also find a proof in its automatic mode which in particular includes an automatic setting of the ordering and selection function. Sometimes a more clever setting of meta-logical directives is required. A more detailed discussion of this issue is provided below (Section 7.2).

2.1.2 Saturation of a Theory

When we are interested in the finite saturation of a theory, we don't expect to find a contradiction and hence want to limit forward computation as much as possible. In that case certain meta-logical directives, including the reduction ordering, have to be set manually (Section 7.3).

2.1.3 Establishing Complexity Bounds

This is a new application of saturation up to redundancy by ordered resolution. For a finitely presented saturated theory, depending on the ordering that was used for saturation, we may obtain a complexity result for the entailment problem for that theory. We call a theory local if it can be saturated up to redundancy for an ordering with a specific complexity bound. The setting of directives for establishing locality is discussed below (Section 7.4). Specific orderings must be used and the full power with respect to redundancy proofs has to be requested.

2.2 Inference System

Saturate applies two kinds of inferences, chaining inferences based on known transitivity properties for binary relations (including the transitivity of logical implication and equivalence), and certain rules derived from sequent calculus for eliminating logical symbols and quantifiers.

2.2.1 Chaining Inferences

Transitivity of a relation R, say, is dealt with by *chaining* inferences of the form

$$\frac{C,\ R(s,t) \qquad D,\ R(u,v)}{(C,\ D,\ R(s,v))\rho}$$

with ρ the mgu of t and u. Its dual, *negative chaining*, is an inference of the form

$$\frac{C,\ R(t,s) \quad D,\ \neg R(u,v)}{(C,\ D,\ \neg R(s,v))\rho}$$

or

$$\frac{C,\ R(s,t) \quad D,\ \neg R(v,u)}{(C,\ D,\ \neg R(v,s))\rho}$$

with ρ the mgu of t and u.

As can be seen, paramodulation is a special case of chaining. In fact, equality is a binary relation for which we have transitivity, reflexivity, symmetry and monotonicity for all function and predicate symbols. Monotonicity or anti-monotonicity axioms can be dealt with by chaining into subterms. At present, subterm chaining is only implemented for partial congruences including equality. Resolution, too, is a special case of negative chaining

$$\frac{C,\ A == \top \quad D,\ \neg(B == \top)}{C\sigma,\ D\sigma,\ \neg(\top == \top)}$$

between two equivalences $A == \top$ and $B == \top$ with subsequent deletion of the trivial equation $\top == \top$, where ρ is the mgu of A and B.

Saturate attempts to automatically recognize relevant properties of binary relations (Section 2.2.4), including transitivity, in order to find an appropriate tailoring of the chaining inference system.

The chaining inference system in Saturate is parameterized by essentially two meta-logical parameters: orderings (Section 2.2.1) and selection function (Section 2.2.1). Ordering and selection function induce essential restrictions (Section 2.2.1) that one may place on inferences without compromising refutational completeness.

Reduction Orderings. When we speak of an ordering we mean a total (or completable) reduction ordering. A reduction ordering is a well-founded ordering on ground terms that is compatible with contexts and is total or can be extended to a total such ordering. Reduction orderings are the basis for terminating computations by term rewriting. Saturate provides several classes of reduction orderings (Section 6.1.3), including the lexicographic path ordering and its AC-variant. Orderings specify an orientation of binary atoms, in particular equalities, indicating the preferred direction when replacing equals by equals. A ordering $\succ$ also expresses preferences and directions of search when attempting to construct models for disjunctions: $A \vee B$ is to be viewed as $\neg B \rightarrow A$, if $A \succ B$, and this links ordered resolution to negation as failure in logic programming.

Selection Functions. With the concept of selection functions one exploits the don't-care nondeterminism that comes with negative literals in clauses: To refute $A_1, \ldots, A_k \rightarrow$ requires to refute each of the A_i, but the order in which this is done is irrelevant. To eliminate redundancy that might arise from this nondeterminism, selection functions that select negative literals can be defined with the command `sass` (Section 6.4.6).

Restrictions for Inferences. Given an ordering and a selection function, the chaining inferences are restricted in that an inference is performed only between maximal or selected atoms in a clause, and only if the middle term through which is being chained is maximal (with respect to the given reduction ordering $\succ$) in both atoms:

1. $s\sigma \not\preceq t\sigma$,
2. $v\sigma \not\preceq u\sigma$,
3. the participating R-atoms must either be maximal or selected within the repsective premises; and
4. the positive (that is, first) premise of an inference contains no selected literals.

Hence, selection not only indicates which subgoal to solve in a clause. It also serves to block clauses from becoming a positive premise in an inference.

Unrestricted chaining amounts to hyperresolution with the transitivity axiom and creates a large search space. This is illustrated by the two examples `HRES15`[2] and `OC15`[3] which both consist of the facts $i > i+1$, for $i = 1, \ldots, 14$, together with the transitivity axiom for $>$. In the case of `HRES15`, the ordering restrictions are turned off. To saturate `HRES15` takes

[2] File `hres15` in the examples directory of the Saturate distribution
[3] File `oc15` in the examples directory of the Saturate distribution

about 100 times longer than running the example in the unsual ordered chaining mode `OC15`.

2.2.2 Sequent Calculus

Sequent calculus is usually viewed as being applied in a backward manner to construct proofs for sequents S by eliminating the logical symbols in S one after the other. For this mode of application cuts are not required, although their use might lead to much smaller proofs.

Saturation-based theorem proving can be viewed as a mechanism for establishing

$$S_1, \ldots, S_k \vdash \rightarrow$$

that is, for deriving the empty sequent $\rightarrow$ from an inconsistent set $S_1, \ldots, S_k$ of sequents, by the *forward application* of the rules of the sequent calculus. In that context, the cut rule becomes the main working horse, and its combination with the substitution rule, which is an inverse form of certain quantifier rules, is called resolution. That view of resolution allows one to integrate clausal normalform transformation into resolution and chaining in a lazy manner.

The (multi-conclusion) rules that we apply in a forward manner are the inverse forms

$$\frac{\Gamma \rightarrow F \wedge G, \Delta}{\Gamma \rightarrow F, \Delta \quad \Gamma \rightarrow G, \Delta}$$

of invertible rules such as

$$\frac{\Gamma \rightarrow F, \Delta \quad \Gamma \rightarrow G, \Delta}{\Gamma \rightarrow F \wedge G, \Delta}$$

for the logical connectives. (The expanded formula $F \wedge G$ of the inference is called its *main formula* below.) The quantifier rules that we use are

$$\frac{\Gamma \rightarrow \forall x.F(x),\ \Delta}{\Gamma \rightarrow F(X),\ \Delta}$$

where X is a fresh free variable, and

$$\frac{\Gamma \rightarrow \exists x.F(x, \vec{Y}),\ \Delta}{\Gamma \rightarrow F(f(\vec{Y}), \vec{Y}),\ \Delta}$$

where f is a new (Skolem) function symbol and $\vec{Y}$ are the free variables in F. (The "left" versions of the rules are analoguous.)

Note that these rules are simplifications in that the premise can be deleted once its conclusions have been derived. In fact, the conclusions imply their premise and are smaller in an appropriate extension of the atom ordering to general first-order formulas. However, we apply rules only if the main formula is either maximal in the sequent or negative and selected. In other words, the rules obey the same restrictions with respect to ordering and selection functions as the chaining inferences.

Sequents in Saturate may contain equalities $F == G$ between formulas F and G, semantically denoting equivalence of F and G, which are dealt with specifically. Only if a maximal one of the two sides F and G is non-atomic, the equality is expanded according to

$$\frac{\Gamma \rightarrow F == G, \Delta}{\Gamma, F \rightarrow G, \Delta \quad \Gamma, G \rightarrow F, \Delta}$$

In case $F \succ G$ and F is atomic, i.e., of the form $p(s_1, \ldots, s_k)$, with a predicate symbol p and terms s_i, the equivalence is *not* eliminated from the sequent. Non-eliminated equivalences are subject to ordering-restricted forms of chaining inferences such as

$$\frac{\Gamma \rightarrow A == F, \Delta \quad \Lambda \rightarrow B == G, \Pi}{(\Gamma, \Lambda \rightarrow F == G, \Delta, \Pi)\sigma}$$

with σ the mgu of the atoms A and B, provided $F\sigma \not\succeq A\sigma$, $G\sigma \not\succeq A\sigma$, no premise contains a selected literal, and both equivalences are maximal in their respective premises.

Orientable equivalences $A == F$ in which the maximal side A is atomic are also used for simplification in that everywhere occurences of $A\sigma$ will be replaced by $F\sigma$. This handling of equivalences is especially useful for definition hierarchies if the definitions are to be applied in a goal to replace derived predicates and if one does not want to automatically infer lemmas for the new predicates. An example how proofs in Saturate look like when simplification with equivalences are intertwined with chaining is a simple set theory[4] with an ordering[5] under which definitions are expanded out.

2.2.3 Modal Logics

Saturate includes a translator for modal logic into first-order logic by semantic embedding. The translator will be implicitly invoked when calling `prove` (Section 6.11) for a modal logic formula.

2.2.4 Specific Treatment of Binary Relations

Saturate attempts to automatically recognize certain properties of binary relations and then specializes the chaining-based inference rules accordingly. Also, the simplification module will know about these properties. The system recognizes and treats in a specific manner

[4]File `set` in the examples directory of the Saturate distribution

[5]File `set.ord` in the examples directory of the Saturate distribution

transitivity or, more generally, certain collections of composition laws of the form

$$p(X,Y), q(Y,Z) \to r(X,Z),$$

for binary relations p, q, and r;

totality laws of the form

$$\to p(X,Y),\ p(Y,X),\ X = Y\ ;$$

reflexivity and irreflexivity laws of the form

$$\to p(X,X)$$

or

$$p(X,X) \to\ ;$$

monotonicity laws of the form

$$p(X,Y) \to p(f(\ldots X \ldots), f(\ldots Y \ldots))\ ;$$

anti-monotonicity laws of the form

$$p(X,Y) \to p(f(\ldots Y \ldots), f(\ldots X \ldots))\ ;$$

symmetry laws of the form

$$\to p(X,Y),\ p(Y,X)\ ;$$

density laws of the form

$$p(X,Y) \to p(X, d(X,Y)) \wedge p(d(X,Y), Y)$$

(written as two clauses); and

laws about the absense of endpoints of the form

$$\to p(X, f(X))$$

or

$$\to p(g(X), X).$$

Interesting types of binary relations with such properties include equivalences, equality (congruences) and orderings.

2.3 Redundancy

Proving the redundancy of clauses and inferences is indispensible for pruning the search space non-locally. With the techniques that are implemented in Saturate we can effectively compute saturated sets of clauses in more cases than with most other saturation-based provers. Finite, saturated sets of clauses not containing the empty clause are provably consistent, and more efficient refutational theorem proving strategies become available for proving consequences of such theories. Sometimes they provide the basis for a decision procedure for the respective theory.

Redundany is closely tied to the term ordering that one uses. In short, a clause is redundant in a particular state of the saturation process if it is logically entailed by smaller clauses that exist in that state. For inferences a similar criterion applies: if the conclusion is entailed by clauses smaller than the negative premise it is redundant (see (Bachmair & Ganzinger 1994*b*). Ordering constraints as they arise from inferences can be kept with the conclusion and inherited to its descendants as described in (Nieuwenhuis & Rubio 1992*a*) and (Nieuwenhuis & Rubio 1992*b*). This technique not only provides for a more precise checking of the constraints, it also increases the power in checking for redundancy of inferences.

2.4 Associative-Commutative Operators

Saturate provides two ways of dealing with AC-operators. One is by the standard methods of ordered rewriting. This is in many cases not good enough. On problems in which AC-symbols play a central role and interact non-trivially with some transitive relation, ordered rewriting as provided in Saturate may perform poorly, although AC-specific redundancy proof techniques (e.g., convergence tests for critical pairs by case analysis and constrained rewriting) are provided. A non-trivial case in which ordered rewriting together with these techniques works well is an exampe due to Overbeek[6]. In that example, saturation not only finds the required commutativity property of multiplication but moreover terminates with a ground convergent system for the commutative ring in question.

Alternatively, and this is the standard setting, Saturate will apply AC-superposition and AC-chaining based on AC-unification, extended clauses and AC-orderings as explained in (Bachmair & Ganzinger 1995*a*). AC-unification can be very expensive. Most of the time, additional algebraic structure e.g., cancellativity, group or ring structure, is present and, hence, should be exploited. This is one of the directions into which future research concentrates on.

[6]File `overbeek3` in the examples directory of the Saturate distribution

3 Getting started

The system is implemented in Prolog and expects commands in the syntactic form of Prolog goals. The list of all the Saturate commands is given in Section 6. For proving Φ in automatic mode enter `prove(`Φ`)`. The syntax of formulas is explained in Section 4.

In semi-automatic mode, or for saturating a consistent theory, the typical interaction cycle (Section 5) with Saturate consists of reading a set of (general) clauses, issuing commands to define the reduction ordering, inference system, redundancy checks, system messages and profiling flags, and, finally, starting the saturation process. If saturation terminates or is interrupted (by typing control-C), proofs of generated clauses can be inspected. Saturated systems of clauses can be filed out for later uses as modules of more complex theories.

4 Formula Syntax

First-order formulas are built in from

- Prolog variables, denoting free or bound variables,
- logical symbols `=`, `~`, `and`, `or`, `<=>`, `=>`, `all`, and `exists`, denoting equality, negation, conjunction, disjunction, equivalence, implication, universal and existential quantification, respectively, and
- Prolog terms for denoting terms and atoms.

General or standard first-order *clauses* are written as sequents Γ `->` Δ of multisets Γ (the *antecedent*) and Δ (the *succedent*) of general literals. The elements in Γ and Δ are separated by ",". A *general literal* is either a first-order formulas F or, somewhat unusually, an *equality* F `==` G *between two formulas* F *and* G. As usual, comma on the right side of `->` denotes disjunction, and comma left denotes conjunction. Equality on the level of formulas denotes logical equivalence. This is an example of a sequent consisting of a single (and positive) equality:

```
-> eq(X,Y) == all(Z,member(Z,X) <=> member(Z,Y))
```

Free variables in a clause, such as X and Y above, are implicitly universally quantified. Here is an example of a (Horn) sequent

```
exists(X,f(X)), exists(X,g(X)) ->
      all(X,(f(X) => h(X)) and all(X,g(X)=>j(X)))
                           ==
      all(X,all(Y,f(X) and g(Y) => h(X) and j(Y)))
```

with a 2-element antecedent and an ==-equality as succedent. We have an equivalence `==` on the level of sequents (in addition to the equivalence `<=>` on the level of formulas) as an explicit indication for our computing with certain such equivalences by rewriting. A detailed definition of the syntax is given below (Section 4).

Context-free Syntax for Clauses. Nonterminals in the subsequent grammar are the symbols that begin with an upper-case letter.

```
Clause  : ProperClause
        | goal ProperClause

ProperClause : Antecedent -> Succedent.  -- sequent notation
             | Succedent <- Antecedent.  -- reverse sequents
             | Succedent.                -- positive disjunction
             | axiom( [ AntecedentL, SuccedentL, Constraint ] ).
                                         -- clauses with initial constraints
             | Literals.                 -- Otter-like notation

Antecedent : Multiset
Succedent  : []                          -- empty succedent
           | Multiset
Multiset   : Formula ‘‘,’’ Multiset | Formula

AntecedentL, SuccedentL: List(Formula)

Literals : Literal OrSymbol Literals
         | Literal

Literal  : Formula                       -- positive literal
         | - Formula                     -- negative literal

OrSymbol : ‘‘;’’ | ‘‘|’’


Formula  : Form
         | Form == Form                  -- equivalence
Form     : all(Variable,Form)
         | exists(Variable,Form)
         | Form Op Form
         | ~ Form
         | Atom
Op       : and | or | => | <=>
```

```
Atom      : Eq
          | NonEqAtom

Eq        : Term=Term                      -- term equation
          | NonEqAtom
NonEqAtom: p(Term,...,Term)    where p is a predicate symbol


Constraint: list( Conjuncts )              -- interpreted as a dnf
Conjunct  : [ List(Ineq) , List(Eq) ]
Ineq      : Term > Term | Term >= Term

Term      : Prolog term
Variable  : Prolog variable
List(X)   : Prolog list with elements of type X
```

In addition, the usual Prolog operator precedences apply, plus the following precedences

```
:- op(920,yfx,[and]).
:- op(930,yfx,[or]).
:- op(950,xfx,[<=>]).
:- op(950,xfy,[=>]).
:- op(900,fy,[~,dia,box]).
:- op(910,yfx,[::]).
:- op(910,xfx,[^^]).
```

(The symbols `dia`, `box`, `::`, and `^^` are used for denoting modal logic operators (Section 2.2.3) in connection with `translate` (Section 6.14).)

Clauses may have ordering and equality constraints attached. (In constraints only term equations are admitted.) In that case a clause represents all the ground instances which satisfy the constraint. The constraints are interpreted over the selected reduction ordering. The ordering is selected by an optional ordering declaration (Section 6.1.3). The constraints mainly serve to reduce the search space for the saturation process and to delay immediate constraint solving upon inference computation. If initial input clauses have constraints refutational theorem proving might be incomplete.

Distinguishing goal clauses (prefix `goal`) from theory clauses may help in finding proofs faster. Inferences in which at least one premise is derived from a goal clause are selected with higher priority (without violating fairness, however).

There is one important side condition that we make but don't check: `[]` and `[_|_]` should not be used as function or predicate symbols. Also, certain symbols, in particular "not" and ":", that have a meaning on the level of atoms or goals in Prolog programs should not be used. *Very* strange thing may happen, otherwise.

5 The Interaction Cycle

5.1 Reading Clauses

The commands `in` (Section 6.6) or `add` (Section 6.1) read files containing clauses and meta-logical directives. We support structured theories, the separation between logical and meta-logical items and the use of pre-saturated theory modules. The clauses in the database can have different origins. The command `l` (Section 6.7) displays the current saturation status of the current theory.

5.2 Selecting the Inference System

Before starting a saturation, the user defines the various parameters of the saturation process. All commands that set some system parameter start with the prefix `sa` for "set active ... ". The system comes with default settings that should be fine for initial experimentation. Parameter settings are also predefined specifically for each example directory (cf. `cd` (Section 6.2)) and may be included in problem definitions (cf. `in` (Section 6.6)).

The inference system is set with the commands `sair` (Section 6.4.3) and `saes` (Section 6.4.2) The inference system is further parameterized by a selection strategy for negative literals. This strategy is (re-)defined with the command `sass` (Section 6.4.6).

5.3 Selecting Ordering and Selection Function

Orderings are defined with ordering declarations (Section 6.1.3) in a file or, interactively, with the `prec` (Section 6.10) command.

5.4 Selecting the redundancy proof techniques

This is done with the command `sarp` (Section 6.4.5). Redundancy proofs allow to

- simplify clauses, including
  - simplification by (contextual) reduction (demodulation),
  - simplification by totality resolution,
  - condensement,
  - applying specific propositional techniques to sets of ground clauses, and

    - simplification by applying "noncritical" tableau expansions;
- delete redundant clauses by
    - tautology detection,
    - subsumption,
    - case analysis over all possible orderings of variables, and
    - clausal rewriting.

More information on these techniques is contained in (Bachmair & Ganzinger 1994*b*) and a paper by Nivela and Nieuwenhuis.[7]

5.5 Selecting the output information

The command `sams` (Section 6.4.4) allows to control the output information that is presented to the user during a saturation proccess.

5.6 Selecting timing points

The command `satp` (Section 6.4.7) allows to profile execution times for the main saturation modules.

5.7 Saturating a system

The command `sat` (Section 6.13) saturates the current data base. When the saturation is finished, the the final set of axioms is displayed, together with the proof, if the empty clause was generated. Saturation can be forced to terminate with a partially saturated system by providing an upper bound on clause generation.

5.8 Saving (partially) saturated set

The command `out` (Section 6.9) dumps the current theory in its current saturation state onto a specified file. Under that name the theory may be included as a module into other theories. In that case, any subsequent saturation for the enriched thepry will not compute any inferences between theory module clauses. That is incomplete in case the module was not fully saturated at the tome when it was dumped.

5.9 Proof Printing

The command `rp` (Section 6.12) displays the proof of the empty clause. The proof for every other clause with number N in the data base can be obtained by the command `rp`(N).

[7] This paper is provided with the `Saturate` distribution in the `papers` subdirectory.

6 Saturate Commands

6.1 add: Adding Input

$$\texttt{add}(I),\ \texttt{addsrc}(I)$$

read input file(s) specified by I and adds their contents to the current theory.

Input files define a first-order theory given in the form of general clauses. Input files may also specify a reduction ordering and other meta-logical directives that serve to guide the saturation process. Before reading the details of this section, you may want to look at a few examples (Section 6.1.8) of input files first.

Saturate (from version 2.4 onwards) also accepts input files in Otter format or in TPTP format (with `include` declarations and `input-clause` facts). From version 2.5 onwards clauses may consist of *general literals* which are arbitrary, signed first-order formulas, possibly with quantifiers.

More precisely, input files consist of sequences of clauses (Section 4), together with optional

- use declarations (Section 6.1.1)
- predicate declarations (Section 6.1.2)
- ordering declarations (Section 6.1.3)
- precedence declarations (Section 6.1.4)
- outputs from `out` (Section 6.9) commands (Section 6.1.5)
- flag settings (Section 6.1.6)
- option settings. (Section 6.1.7)

Clauses, or theories and problems, rather, may also be specified in the form of DFG-Syntax. For an example see `Pelletier 24`[8] in DFG-notation.

Following the conventions for Prolog terms, each item must be terminated by a dot "." . Terms and lists of items are to be written in Prolog notation. Comments are written as in Prolog, that is, a single line of comment is indicated by a leading "%" character. Other comments are included in "/*" and "*/" brackets.

We suggest that meta-logical directives be kept separate from theory presentations. How to achieve such separation is described below (Section 6.1.9).

[8] File `pell24` in the examples directory of the Saturate distribution

6.1.1 Use Declarations

Use declarations are of the form `use(`F`)`, where F is an atom denoting a theory to be included. The meaning of that declaration is equivalent to the execution of the command `add(`F`)`.

6.1.2 Predicate Declarations

Predicates that occur in axioms may (but need not) be declared. Only one predicates declaration is allowed in a file.

`predicates(list(PredicateSymbols)).`

In addition, if $R1$ is a binary predicate, by writing

`reflexive_closure(`$R1$`,`$R2$`).`

one declares $R2$ to be the reflexive closure of $R1$. An explicit declaration of the reflexive closure is a shorthand for writing explicitly axioms of the form

$$R1(X,Y) \texttt{ -> } R2(X,Y).$$
$$R2(X,Y) \texttt{ -> } R1(X,Y),\ X = Y.$$

6.1.3 Ordering Declarations

By an optional declaration of the form

`ordering(`Ord`).`

one specifies the (kind of the) termination ordering to be used for ordered inferences and for redundancy proofs. Ord may be either

- `lpo`, for the lexicographic path ordering which is the default setting, or
- `aclpo` for the AC-lexicographic path ordering which is the default setting when AC-operators are present,
- `sto` for a precedence-based subterm ordering, or
- `poly` for (linear) polynomial interpretations[9]

The `sto` ordering is the union of the subterm ordering with inequations $s = f(\ldots) > t = g(\ldots t_i \ldots)$ such that any of the t_i are subterms of s and $f > g$ in the precedence. The atom ordering is a certain lexicographic extension of this term ordering. *sto* is useful, probably, only in the non-equational case when one tries to prove locality of an inference system, cf. ocality and saturation.

[9] Only with the SICStus 3.0 version of Saturate .

6.1.4 Precedence Declarations

All orderings that one may select require a complete precedence of function and predicate symbols. This precedence may be partially or completely defined by the user. Partial precedences will be completed automatically by the system.

The optional declaration of the form

```
top_predicates_precedence(list(PredicateSymbols)).
```

lists all the predicates (in deceasing precedence) that are considered to be greater than any function symbol (including those that occur in subsequent enrichments of this theory) and greater than any other predicate symbol. (In previous versions of Saturate the term `first_predicate_precedence` was used.) The declaration

```
precedence(list(FunctionAndPredicateSymbols)).
```

specifies the possibly partial precedence of the *non-top* predicate and function symbols in decreasing order. There is a pre-defined constant `'$min'` which is smaller than any other symbol. `'$min'` may occur in clauses but must not occur in the precedence declaration. We have examples in which saturation is much faster if the smallest constant is employed (e.g., modal logic KD45[10]).

Each theory module may have separate [top predicates] precedence declarations associated with it in which the symbols of the module are related to each other. When combining modules by importing them into another module, the union of the precedences must be non-circular. In particular it is not allowed to declare a predicate as a "top predicate" in one module and at the same time include it in a precedence declaration for non-top symbols in another module. Precedence declarations in a module which imports one or several modules must be consistent with the declarations in the module(s).

The automatic completion of the precedence seems to work well in most cases for predicate symbols. For function symbols that occur in equations some rather simple pragmatic assumptions are made. In particular, the system attemps to define the precedence such that equations are oriented from left to right.

6.1.5 Reading Outputs from `out`

In this case, saturated sets of clauses are recognized as such, and corresponding inferences will not be recomputed.

[10] File `modal-logic/kd45` in the examples directory of the Saturate distribution

6.1.6 Calls to Flag Settings

Calls to flag settings are of the form

`:- flag(`S`).`

where

```
flag    : sair | saes | saci | sass | sarp |sams | satp
```

The setting S is flag-specific and explained for the respective `sa` (Section 6.4)<xx> command,

Before reading a file, flags are set to the defaults as given by the defaults file in the current directory. If no such file exists it is created with a certain standard set of flag settings. If flag settings should not be reset to their defaults, use `inc(`F`)` rather than `in(`F`)`.

6.1.7 Options Settings

Option settings are of the form

```
:- option(list(O)).
```

where

```
O : term_size_weight | term_var_weight | term_var_weight_gt_small |
    normal_term_weight | increment(N) | var_depth(N) |
    special_relations(on) | special_relations(off) |
    ac(on) | ac(off) |
    memo_constraints(on) | memo_constraints(off)
```

The weight options set the weight function for terms and clauses which is used for inference selection.

`term_size_weight` :

weight = number of nodes in a term

`term_var_weight` weight = weighted product of term size, term depth and number of different variables

`term_var_weight_gt_small` similar, except that ground terms are made very small

`normal_term_weight` (default)

weight = weighted product of term size and depth `increment(`N`)` Inferences are computed from clauses up to a certain weight. If no proof is found, the bound is increased by N. Defaults: $N = 15$ for `term_size_weight`, `normal_term_weight`, $N = 30$, otherwise.

`var_depth(N)` Clauses are preprocessed into rules in an order that also depends on the number of different variables occuring in it. If this number exceeds N (default value is 3), processing of the clause is delayed.

`special_relations(`S`)` $S =$ `on` turns on the special treatment of transitive relations by chaining inferences; default. In this mode, transitivity, reflexivity, irreflexivity totality, density and no endpoint axioms are treated specifically. Saturate recognizes these properties automatically, provided they are not hidden, i.e., follow in a non-trivial way from other properties. Actually Saturate should be more clever in recognizing these properties.

$S =$ `off`: transitive relations are treated as any other predicate. This option should be specified textually before any axioms for the respective predicates. This option essentially reduces the inference system to ordered resolution. It should be used when Saturate is applied in order to prove the ¡locality of a Horn theory.

`ac(S)` $S =$ `on` (default) results in employing AC-superposition based on AC-unification and extended rules for AC symbols.

$S =$ `off` results in the treatment of AC-symbols by ordered rewriting.This option must textually preceed any AC laws.

`memo_constraints(S)` $S =$ `on` switches memoization for ordering constraints on. This is the recommended setting for cases in which big ordering constraints are generated. This is the case in particular when clausal rewriting is turned on.

$S =$ `off` (default): no memoization

`option/1` may also be called interactively.

6.1.8 Examples

For examples browse the `examples` directory of the Saturate distribution.

6.1.9 Separating Logic and Meta-Logical Directives

We suggest (but do not strictly require) that ordering declarations and other meta-logical directives be kept separate from theory presentations. In the `add` and `addsrc` commands I must be an atom F, or a pair `[`FO, F`]` of atoms, representing one or two file names, respectively. In the second form of the command, FO is supposed to specify the name of the file containing the meta-logical directives, while F is supposed to denote the file containing the theory. When using the first form of the command, the system checks for the presence of a file named F`.ord`. If a file by that name exists, it is assumed that it contains relevant meta-logical information associated with F. In that case, `in(`F`)` is equivalent to `in([`F`.ord,`F`])`.

Theories may exist in different presentations. In particular it is possible to dump a (partially) saturated theory module to a file and read it back in later (cf. `out` (Section 6.9) command). In that case inferences will not be (re-)computed if each premise belongs to the (same) theory module. For fully saturated theories such a linear strategy is complete. For partially completed theories it may be incomplete but nevertheless be be useful as a strategy. It is assumed that files with names of the form F`.o` are dumped, partially saturated theory modules. If a file by that name exists, `in(`F`)` will read F`.o` rather than F itself. In cases where this default behaviour is not wanted, `addsrc(`F`)` should be used, which is otherwise equivalent to `in(`F`)`.

6.2 `cd`

`cd(`D`)`

Changes current working directory to D, that is, D becomes the working directory for any subsequent input and output.

The default settings for flags for this directory become the current flag settings and the active defaults for subsequent changes (sa?? commands). If the directory does not contain a defaults specification it will be created in some predefined way and stored in a file (named `defaults`) in that directory. Changes to these defaults can be made subsequently by the `set active ...` commands together with the `save active ...` commands.

D must be a Prolog atom denoting the directory. Sorry, no abbreviations allowed. Write single quotes " ' " around D, if D contains delimiters such as " . " or " / " such as in `cd('/HG/usr/hg/saturate/examples')`.

6.3 `cl`: Clearing the Database

Clears the data base. This is done automatically when calling `in` (Section 6.6).

6.4 Flag Settings

The flag setting commands are named `sa`<xx>, for "Set Active xx" where xx is a two-letter identifier for a flag. The commands can either be called without a parameter for an interactive setting of flags, or with a single parameter S which is a list of options for that flag. Calls to `sa`-commands with parameter may also appear in input files (Section 6.1.6). Options are given by natural numbers. Their meaning is flag-specific and is described below. If the list is prefixed by a "`+`", the listed options are added to the set of current options for the flag. If the list is prefixed by a "`-`" they are deleted.

All flags have a default setting that is determined by a file named `defaults` in the current directory. `Saturate` will generate such a defaults file upon changing into a directory with `cd` (Section 6.2) where there is no defaults file yet. The defaults file can be changed into user-specified defaults with the family of `sv` (Section 6.4.8) commands.

6.4.1 `saci`: Settings for Constraint Inheritance

```
saci
```

specify the setting of the constraint Inheritance strategy. The (pairwise exclusive) options O for constraint inheritance are

$O = 1$: no constraint inheritance but complete immediate checking (default),

$O = 2$: hereditary ordering constraint,

$O = 3$: hereditary equality constraint: basic strategies.

With option 1 one gets a complete checking of ordering constraints with each inference.

With option 2, ordering constaints are inherited from the premises to the conclusion of an inference, enriched by the ordering constraint for that inference. In this case ordering constraints may become prohibitively large. Constraints are only inherited for non-unit clauses. No constraint weakening is performed. That is, even when C subsumes D but the constraint of D does not imply the constraint of C, D will not be deleted. An alternative would be to delete D, and at the same time weaken the constraint for C such that the weaker constraint is implied.

Option 3 is incompletely implemented and should not be used at present.

If all options are turned off by calling `saci([])` clauses are preoriented once and for all, and inferences are computed with maximal literals and terms according to this preorientation. This corresponds to checking ordering constraints before the unfier is applied. If big clauses are produced during saturation, solving ordering constraints may be prohibitively expensive and `saci([])` might be a good choice.

6.4.2 `saes`: Settings for Eager Simplification

```
saes
```

The saes-flag specifies the list of simplification rules S that are to be applied eagerly on each clause so as to ensure refutational completeness of a certain inference system. At present we have only one such inference, called "variable elimination', and denoted by $S = 1$, which should usually be turned on.

Variable elimination for orderings in Saturate is explained in (Bachmair & Ganzinger 1994*a*). Variable elimination also includes splitting of variable-disjoint parts of clauses or variable-disjoint sides of equations.

6.4.3 sair: Settings for Inference Rules

`sair`

This flag lists the set of inference rules R that are to be applied fairly to persisting clauses.

The rules are identified by numbers as follows:

$R = 1$: reflexivity resolution

$R = 2$: right superposition and chaining

$R = 3$: left superposition and negative chaining

$R = 4$: factoring

$R = 5$: ordered resolution

$R = 6$: transitivity resolution

$R = 7$: tableau expansion

The inference system consisting of all theses rules is refutationally complete In the case of dense orderings, one additionally needs to switch on variable elimination for refutational completeness, cf. command saes (Section 6.4.2). The default setting is determined by the current theory and assures that sufficiently many inference rules are turned on. Transitivity resolution is only needed for specific cases of transitive relations.

It is a good strategy (with respect to time efficiency) to set active inferences according to the problem at hand; for instance, for a purely equational problem only inference 3 (right superposition) is needed.

The inference system can be further instantiated with strategies for

1. constraint inheritance: command saci (Section 6.4.1)
2. selecting negative literals: command sass (Section 6.4.6), and
3. eager simplifications: command saes (Section 6.4.2).

For a description of these inferences we refer to our overview (Section 2.2) and to (Bachmair & Ganzinger 1995*b*).

6.4.4 `sams`: Messages

`sams`

settings for the printing of MeSsages during saturation. Find out about the flags interactively by calling `sams` without parameters.

6.4.5 `sarp`: Selecting Redundancy Proof Techniques

`sarp`

specifies the set of techniques for Redundancy Proofs and for simplification.

These methods are currently implemented and associated with these options P:

$P = 1$: tautology elimination

$P = 2$: forward subsumption

$P = 3$: backward subsumption

$P = 4$: condensement

$P = 5$: propositional reasoning for ground clauses

$P = 6$: forward simplification by reduction

$P = 7$: backward simplification by reduction

$P = 8$: redundancy tests by case analysis (forward)

$P = 9$: redundancy tests by case analysis (backward)

$P = 10$: redundancy tests by clausal rewriting (forward)

$P = 12$: totality resolution

$P = 13$: extensionality resolution

$P = 14$: minimal level of reduction

$P = 15$: maximal level of reduction

If $P = 15$ is set, reduction is *recursive rewriting* of "maximal strength" where most clauses and their contrapositives are considered as conditional rewrite rules. Reduction also includes *convergence tests* for inequalities in clauses with respect to the current set of positive unit inequalities. This setting can be very expensive. With $P = 14$, only unit clauses are used for reduction which usually is much faster. If neither option is present, an intermediate level of reduction strength is assumed (default setting).

Subsumption always includes a particular form of contextual reduction, also called *resolution subsumption*. Consider a resolution inference

$$\frac{C, A \quad D, B}{D}$$

in the case when $A\sigma = \neg B$ and $C\sigma \subseteq D$. Here the conclusion subsumes the second premise which can then be deleted. As to whether such an inference is possible is detected by a subsumption-like algorithm. This is why it has been included into subsumption so as to not duplicate work. Reductions of this form are extremely useful in many examples.

Propositional reasoning for ground clauses refers to an integration of an incremental Davis-Putnam-like procedure for testing satisfiablity of propositional clauses contributed by Peter Barth. The procedure is part of his CLP(PB)-system and is operational only in the SICStus 3.0 version of Saturate.Refutational completeness with this method switched on is guaranteed only for ordered strategies without selection as invoked through `saci([])` (Section 6.4.1).

Totality resolution eliminates, in the case of total relations p, all negative occurrences of p by swapping them to the other side of the sequent. This option, by contrast to using the totality axiom lazily, is generally recommended for total orderings, in particular when combined with a positive strategy (`sass([2])` (Section 6.4.6)).

Extensionality resolution is a simplification related to the presence of an extensionality axiom for equality or some other transitive relation. It is somewhat experimental in nature.

Case analysis and clausal rewriting are described in more detail in paper by Nivela and Nieuwenhuis. Case analysis is useful for theories with AC-operators that are to be dealt with by ordered rewriting with `option(ac(off))`. Clausal rewriting should be turned on for non-trivial consistent theories that one wants to saturate finitely. For refutational theorem proving it should be turned off as it is very expensive.

6.4.6 `sass`: Choosing a Selection Strategy

`sass`

specifying the Selection Strategy. Selection has a similar meaning as in SLD-resolution. Selection is, in addition to the reduction ordering, the second main parameter to the chaining and paramodulation inference systems.

One may choose from 4 alternatives. `sass([])` turns selection off, resulting in a fully ordered strategy. `sass([`F`])` specifies one of three selection strategies:

$F = 1$: (default) automatic selection of a maximal literal in non-reductive clauses

$F = 2$: automatic selection of a small literal in non-positive clauses

$F = 3$: user guided literal selection

Option $F = 3$ is the one where the user is asked interactively for each non-positive clause to select a negative literal for chaining and superposition.

In option $F = 1$ which is fully automatic, no negative literal is selected if the clause has a positive literal which is strictly greater than all negative literals. Otherwise one, usually maximal, negative literal is selected.

Option $F = 2$ always selects a (if possible non-maximal) literal in any non-positive clause. This results in a positive strategy.

The system indicates, by printing an asterix "*", those clauses for which a literal has been selected. The selected literal is listed first in the antecedent.

Option $F = 1$ sounds like applying usual ordering constraints to inferences. It is different in that we now perform all negative chainings into the selected literal, even in cases where the substituted literal (after applying the unifier of the inference) is not maximal. What we gain is that we never need to chain *from* that clause. Moreover the ordering constraints are simpler and hence faster to check.

This strategy seems to work better than the fully ordered strategy most of the time. Highly recommended in cases where clauses are too long to admit full checking of ordering constraints.

`sass` has been called `sama` in earlier versions of Saturate.

6.4.7 `satp`: Timing

`satp`

Show/Set the Active Timing Points T for profiling.

Profiling can measure times for

$T = 1$: inference computation

$T = 2$: forward simplification and redundancy tests

$T = 3$: backward simplification and redundancy tests

$T = 4$: constraint solving

$T = 5$: clause preprocessing

$T = 6$: rewriting.

Can be quite expensive.

6.4.8 `sv`: Saving Default Flag Settings

`sv`<xx>`d`

SaVe the setting for flag $xx \in \{\texttt{ci}, \texttt{es}, \texttt{ir}, \texttt{ma}, \texttt{rp}, \texttt{tp}\}$ as default into the `defaults` file in the current directory.

6.5 help

`help`

informs an already running web browser, or calls the web browser that has been specified upon installation, for accessing this manual.

6.6 in, insrc: Reading Input Files

in(I), insrc(I)

Reads input file(s) specified by I. I must be an atom F, or a pair [FO,F] of atoms, specifying one or two file names, respectively, and adds them to the **empty** theory. Except for the fact that the database is cleared before reading, the meaning is the same as for add(I) (Section 6.1) and addsrc(I), respectively.

6.7 l: Listing

Lists the contents of the database. In that listing *new clauses* and *persisting clauses*, sometimes also called *rules*, are distinguished. Rules are clauses that have been preprocessed by the system. The distinction is reminiscent of that between equations and (oriented) rewrite rules.

6.8 lprec: Listing of the Precedence

Lists the current precedence.

6.9 out: Dumping Theories to Files

out(F)

Saves the current theory in its current perhaps only partially saturated form, including the ordering definition, into file F. The saved file can be used as a theory module in `in` and `add` commands as well as in `use` directives, cf. `add` (Section 6.1). If the module is part of a theory that is to be saturated, no inferences will be computed in which all premises are clauses of the theory module.

Warning: even if the theory is not fully saturated upon saving, later reuses of the theory might not consider all inferences between theory clauses.

6.10 prec: Precedence Definition

`prec`

ask for a file f for reading predecence declarations (Section 6.1.4). For f = `user` the predecence declarations may be typed in interactively, terminated by `^D`.

6.11 `prove`: Theorem Proving

`prove(`F`)`

invokes a theorem proving process for F by negating F and saturating $\neg F$. The general form of `prove` is

`prove(`F, L, M, O`)`

for proving F in logic L with translation option M and Saturate options O. L and M are as for the function `translate` (Section 6.14). O is a list of `sa` (Section 6.4) flag settings or option settings (Section 6.1.7).

6.12 `rp`: Displaying Proofs

`rp, rp(`N`)`

Displays the proof of clause N. If no argument is provided, the proof of the empty clause is printed. Fails, if the specified clause does not exist (anymore).

6.13 `sat`: Starting Saturation

`sat, sat(`N`)`

This starts the saturation of the set of clauses/rules in the database according to the chosen strategy. In case a number N is provided, saturation stops after N non-redundant clauses have been generated.

Strategies are defined by calling the `sa` (Section 6.4)<`xx`> commands for setting the various parameters. If no settings are provided explicitly, the directory-specific default settings as contained in the local `defaults` file are assumed.

6.14 `translate`: Translating Modal Logic Formulas

`translate`

invokes the translation of a given modal logic formula into first-order logic by means of various semantic embedding techniques that one may choose from. The results from `translate` may be used as an input to `make_file` for creating saturate input files for the background theory of a logic and/or for a problem.

7 Heuristics for the Definition of Meta-Logical Directives

7.1 General

There are two essential ways in which a user can influence the saturation strategy: orderings and selection. The orderings that are available in Saturate have been described above (Section 6.1.3). For the precedence-based orderings, it is difficult to provide generally useful heuristics for setting up the precedence. In any case the ordering should be such that smaller terms and atoms are closer to a canonical form that we want to approximate. That implies that derived symbols usually should get a higher precedence than the symbols they are derived from. Constants are usually minimal. Often predicates should be maximal and top predicates (Section 6.1.4), in particular when their definitions are to be expanded out such as in certain formulations of set theory[11]. Sometimes it is helpful when Skolem functions have a high precedence. For Skolem functions introduced by the system, they are given a precedence higher than any user function. Generated Skolem functions are ordered according to generation order: later symbols (corresponding to inner quantifiers) are smaller than symbols generated earlier. That is, by choosing the ordering of the variables in a quantifier prefix, a user can take influence on the ordering of the corresponding Skolem functions. An example where this is crucial is a certain lemma for the verification of quick sort[12]. Equations in input formulas should be written in the way that one wants them to be ordered. These heuristics are applied when Saturate computes or completes precedences automatically.

While the ordering essentially guides rewriting and search, the selection strategies specify how negative literals should be treated. The available strategies are explained with the `sass` (Section 6.4.6) command. Again it is hard to say which strategy to apply in general. With the ordering-guided strategies `sass([])` and `sass([1])` simplification by recursive (contextual) rewriting seems to have a better effect. Actually, only clauses in which no negative literal is selected may become a potential simplifier at present. For clauses with a selected negative literal we only check as to whether they subsume (or subsume-reduce) some other clause.

Apart from these two pricipal ways of taking influence on search and simplification, one may choose the way how ordering constraints are handled (`saci` (Section 6.4.1) command). One needs to find the right balance between precision and the fraction of runtime allocated for constraint solving. Constraint solving for the `lpo` is NP-complete, and the present implementation clearly shows that. For the future we would like to improve our implementation by providing better heuristics for constraint simplification.

[11] File `set` in the examples directory of the Saturate distribution

[12] File `hoare/qs2` in the examples directory of the Saturate distribution

The best choice usually is `saci([1])`. Some examples run much better with constraint inheritance `saci([2])`, e.g., this formulation of the theorem of the sum of two continuous functions being continuous[13]. Constraint inheritance might be crucial for the finite saturation of a consistent theory.

Finally, we have a number of flags (Section 6.1.6) and options (Section 6.1.7) that one may set in addition. These, among others, affect the selection of inferences according to certain complexity measures on terms and clauses. Setting them one way or another is hacking, and nothing general can be said about it.

7.2 Settings for Refutational Theorem Proving

For refutational theorem proving, we have to find the right balance between forward computation (search) and simplification. A modest amount of simplification should be applied. A refutationally complete inference system is usually wanted. We get this automatically with the default settings of the system. We either call `prove` (Section 6.11) or else read a set of clauses (command `in` (Section 6.6)) containing the theory and the negation of the theorem to be proved, and then enter `sat` (Section 6.13) to invoke saturation. We recommend to use the lexicographic path ordering (default ordering) for refutational theorem proving. Then the definition of an ordering reduces to defining the precedences between the symbols (cf. `in` (Section 6.6)).

7.3 Saturation of a Theory

When we are interested in the finite saturation of a theory, we don't expect to find a contradiction and hence want to limit forward computation as much as possible. In that case we might want to switch on some of the more complex proof techniques for redundancy such as case analysis or clausal rewriting (cf. `sarp` (Section 6.4.5) command). Case analysis is particularly useful in the presence of AC-symbols (Section 2.4). Clausal rewriting has no effect on purely equational theories. It works best in a resolution context when equality plays no role at all. The positive selection strategy (`sass([2])`) is usually not appropriate in this context. Usually the lexicographic path orderings `lpo` and `aclpo` with an aprropriately defined precedence work best in this context. Once we have managed to saturate a theory finitely, all proofs with respect to that theory are linear in the sense that they have the set-of-support property.

7.4 Establishing Complexity Bounds

Three aspects are of importance when using Saturate to derive complexity bounds for entailment.

[13] File `bledsoe/jar3` in the examples directory of the Saturate distribution

- We need to use an ordering in which ground terms have at most finitely many smaller terms. An lpo is, hence, in most cases not suitable.

- The treatment of transitive relations by chaining has to be turned off (cf. flag settings (Section 6.1.6)) to invoke ordered resolution also for transitive relations and, in particular, for the transitivity axiom itself.

- No selection of negative literals is allowed, cf. command `sass` (Section 6.4.6).

For establishing a polynomial complexity bound, the (precedence-modified) subterm ordering `sto` should be chosen with an appropriate ordering declaration (Section 6.1.3). Finitely saturating a theory under an instance of this class of partial orderings (actually one saturates with respect to all possible well-founded completions of that instance simultaneously) proves that the entailment problem for the respective theory is decidable in polynomial time if the theory is Horn, and in CoNP, if it is not Horn. We refer to (Basin & Ganzinger 1995) for a detailed explanation of the theoretical basis.

The subdirectory `LocalTheories`[14] of `saturate/examples` shows examples that we have saturated with respect to the sto, including the theory of term embedding[15], congruence closure[16], various fragments of propositional calculus[17], and the largest PTIME fragment of Allen's interval logic: the theory of partial nonstrict orderings[18]. A formulation of sequent calculus[19] has been saturated modulo AC and with respect to a linear interpretation function as ordering. The details are explained in the respective file. Term orderings via linear interpretations are available only in the Saturate version under SICStus 3.0. Such interpretations may be useful when one intends to prove exponential upper bounds for the complexity of entailment problems.

Proving the locality of a theory usually requires clausal rewriting with an appropriate setting of `sarp` (Section 6.4.5). in addition to the usual simplification and deletion techniques. In some cases hereditary ordering constraints with `sass([2])` (Section 6.4.6) will be useful.

References

Bachmair, L. & Ganzinger, H. (1994*a*), Ordered chaining for total orderings, *in* 'Proc. 12*th* International Conference on Automated Deduction', Vol. 814 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, pp. 435–450.

[14] File `LocalTheories` in the examples directory of the Saturate distribution

[15] File `LocalTheories/emb` in the examples directory of the Saturate distribution

[16] File `LocalTheories/congHO` in the examples directory of the Saturate distribution

[17] File `LocalTheories/bool` in the examples directory of the Saturate distribution

[18] File `LocalTheories/ord` in the examples directory of the Saturate distribution

[19] File `LocalTheories/lk` in the examples directory of the Saturate distribution

Bachmair, L. & Ganzinger, H. (1994*b*), 'Rewrite-based equational theorem proving with selection and simplification', *J. Logic and Computation* **4**(3), 217–247. Revised version of Technical Report MPI-I-91-208, 1991.

Bachmair, L. & Ganzinger, H. (1994*c*), Rewrite techniques for transitive relations, *in* 'Proc. 9th IEEE Symposium on Logic in Computer Science', IEEE Computer Society Press, pp. 384–393.
*http://www.mpi-sb.mpg.de/~hg/pca.html#LICS94

Bachmair, L. & Ganzinger, H. (1995*a*), Associative-commutative superposition, *in* N. Dershowitz & N. Lindenstrauss, eds, 'Proc. 4th Int'l Workshop on Conditional and Typed Rewrite Systems, Jerusalem', Vol. 968 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 1–14. Revised version of MPI-I-93-267, 1993.

Bachmair, L. & Ganzinger, H. (1995*b*), Ordered chaining calculi for first-order theories of binary relations, Research Report MPI-I-95-2-009, Max-Planck-Institut für Informatik, Saarbrücken, Saarbrücken. Revised version to appear in JACM.
*http://www.mpi-sb.mpg.de/~hg/pra.html#MPI-I-95-2-009

Basin, D. & Ganzinger, H. (1995), Automated complexity analysis based on ordered resolution, Research Report MPI-I-95-2-006, Max-Planck-Institut für Informatik, Saarbrücken, Saarbrücken. Full version of LICS'96 paper.
*http://www.mpi-sb.mpg.de/~hg/pra.html#MPI-I-95-2-006

Basin, D. & Ganzinger, H. (1996), Complexity analysis based on ordered resolution, *in* 'Proc. 11th IEEE Symposium on Logic in Computer Science', IEEE Computer Society Press, pp. 456–465.
*http://www.mpi-sb.mpg.de/~hg/pca.html#LICS96

Nieuwenhuis, R. & Rubio, A. (1992*a*), Basic superposition is complete, *in* 'ESOP'92', Vol. 582 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 371–389.

Nieuwenhuis, R. & Rubio, A. (1992*b*), Theorem proving with ordering constrained clauses, *in* 'Automated Deduction — CADE'11', Vol. 607 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 477–491.

A Release Notes

Version 2.5 (October 1996)

- We admit general clauses with general first-order formulas as literals and integrate a kind of lazy CNF transformation into saturation.

- AC-superposition is now fully implemented. However, the AC-unification and matching module is a rather slow prototype. The AC-LPO is available as the default ordering in the presence of AC-symbols. Constraint solving happens by means of a generic solver for ordering constraints which is necessarily incomplete.
- Saturate can also read input files in DFG notation.

Version 2.4 (April 1996)

- Precedences are generated automatically by some analysis of dependencies between predicate and function symbols. Now Saturate can be run fully automatically on a set of clauses without caring about the ordering. This seems to work well in many cases.
- Saturate can also read input files in TPTP notation.

Version 2.3.1 (November 1995)

- We have added support for hierarchical theories.
- Ordering declarations and directives for saturation may be kept separately from the definition of theories, cf. command `in` (Section 6.6).
- We have integrated a Davis-Putnam-like satisfiability checker contributed by Peter Barth (SICStus 3.0 only). This is useful for medium-sized to large near-propositional problems such as in this example[20].

Version 2.3 (September 1995) A new rewrite module has been added. Contextual simplification is now more powerful than it was before.

Version 2.2 (August 1995) Inheritance for ordering constraints has been implemented, cf. command `saci` (Section 6.4.1).

Version 2.2 (June 1995) Adapted to SICStus 3.0.

Version 2.1 (May 1995) We have added a constraint solver for the (lexicographic extension to atoms of the) subterm ordering. This ordering should be used when trying to apply Saturate for proving the locality of certain theories, (cf. subdirectory `LocalTheories`[21] of `saturate/examples`). Finitely saturating a theory under this (partial) ordering (actually one saturates with respect to an arbitrary completion of that ordering) proves that the entailment problem for the respective theory is decidable in polynomial time if

[20] File `happy/andrews3` in the examples directory of the `Saturate` distribution

[21] File `LocalTheories` in the examples directory of the `Saturate` distribution

the theory is Horn, and in CoNP, if it is not Horn. The paper (Basin & Ganzinger 1996) explains the details.

Version 2.0 (April 1995) In this version we use substitution tree indexing for (forward) reduction and subsumption. This makes the system substantially faster on larger examples.

We have made clausal rewriting more powerful such that more theories can be finitely saturated

We support equivalences between (nonequational) atoms A, B, written A `==` B. Equivalences may also occur on literal positions in clauses, cf. `in` (Section 6.6) command.

Version 1.6 In this version in all kinds of transitive relations are dealt with in a uniform way by chaining inferences.

There is still no full implementation for inheriting constraints.

Subdirectories hoare[22] and bledsoe[23] of `saturate/examples` exercise ordering relations. In `hoare` you find lemmas that occur in the verification of quicksort and binary search algorithms. The directory `bledsoe` contains examples from various papers by Bledsoe, including the intermediate value theorem and the challenge problems stated in a JAR paper from 1990.

Version 1.0 This is (was) the first version of the system for logic with equality.

In this version there are no specific redundancy provers for the cases when equality and/or ordering constraints are inherited. Also for the general case, more powerful redundancy notions are being developed.

Since ordering constraint solving is NP-complete, sometimes for very large constraints (say, more than one screen full) it is not possible to decide satisfiability in a reasonable time. In a future version some time-out mechanism has to be provided for these cases. You can circumvent the problem by setting the constraint inheritance mode to `[]`, cf. command `saci` (Section 6.4.1). In this case, a less complete and less time consuming constraint solving mode is applied.

B Known Bugs

Saturate is a prototype that we use to try out some of our theoretical ideas. Many things should be improved, above all, speed. Certainly, Saturate contains bugs that we haven't found yet. There is no correctness guarantee whatsover that we can provide with this system and no liability can be assumed in any regard.

[22] File `hoare` in the examples directory of the Saturate distribution

[23] File `bledsoe` in the examples directory of the Saturate distribution

The following list describes bugs that are related to certain implementation issues.

Reflexive closure declarations (Section 6.1.2) should be made only for strict total orderings. This is not checked. For other transitive relations it may result in cross incompleteness of the prover.

Certain Prolog-operators, in particular `[_|_]`, "`not`" and "`:`" should not appear as function symbols in clauses. Otherwise many things could go wrong. This is not checked.