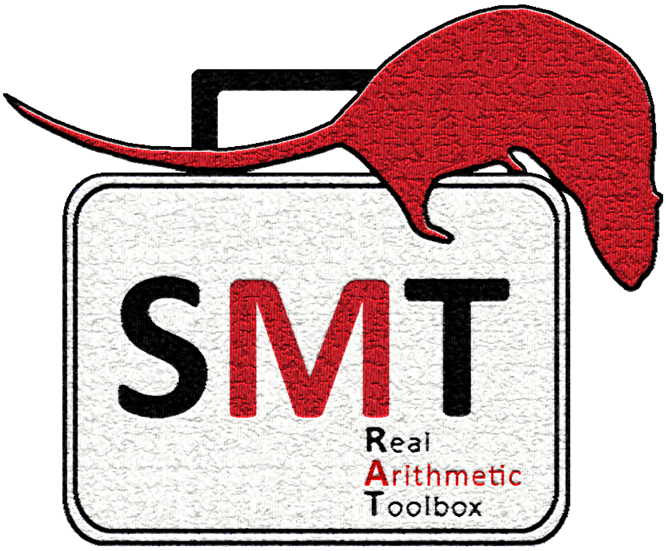
SMT-RAT

Version 2.0.0

Open Source C++ Toolbox
for Strategic and Parallel SMT Solving

Manual



Contents

1 Introduction

This manual describes SMT-RAT, a C++ library consisting of a collection of SMT-compliant implementations of methods for solving nonlinear real and integer arithmetic (NRA/NIA) formulas, we refer to as modules. These modules can be combined to (1) a theory solver in order to extend the supported logics of an existing SMT solver by NRA/NIA or (2) an SMT solver for NRA/NIA. The latter is especially intended to be a testing environment for the development of SMT compliant implementations of further methods tackling NRA/NIA. SMT-RAT provides a graphical user interface for the creation of a strategy of such module combinations. It specifies dynamically which modules have to solve a given NRA/NIA formula, involving the formula's properties and the solving history.

2 Installation

2.1 Requirements

SMT-RAT has been successfully compiled and tested under Linux and Mac OS. For its configuration we use cmake, which can be found on http://www.cmake.org/, and for its compilation we tested successfully gcc (version 4.8 or higher), available on http://gcc.gnu.org/, and clang (version 3.4 or higher). The configuration settings can be changed with the command line interface ccmake. For the arithmetic calculations of rationals SMT-RAT uses the C++ library GMP, which can be found on https://gmplib.org/ (but most often it is already installed on the system). The data structures and basic operations with polynomials and formulas within SMT-RAT are based on the C++ library CArL available at http://ths.informatik.rwth-aachen.de/doxygen/carl/html/. Optional, there is a graphical user interface for the composition of a solver, which needs java (version 1.7 or higher) and its package ANTLR, which can be found on http://www.antlr.org/ .

Summarizing, you need the following packages:

- gcc (version 4.8 or higher) or alternatively clang (version 3.4 or higher)
- cmake
- GMP
- CArL

Optional:

- ccmake [for the command line interface for compiler settings]
- java (version 1.7 or higher) [for the GUI]
- ANTLR [for the GUI]

2.2 Building SMT-RAT

You can download SMT-RAT from https://github.com/smtrat/smtrat and build it the following way:

1. Open a terminal and navigate to the root folder of SMT-RAT.
2. Create a directory (e.g. build) that will contain the object files and the executables, and change into it:

```
mkdir build && cd build
```

3. Configure:

```
cmake ..
```

4. Build:

```
make
```

5. Cleaning:

```
make clean
```

More information can be found in the README file, which can be found in SMT-RAT directory.

2.3 Execute SMT-RAT as an SMT solver

You can find the executable of SMT-RAT, named smtrat, in the build-directory. It displays the usage information if it is invoked with the option flag --help:

```
./smtrat --help
```

The executable supports only .smt2-files as input, so the solving can be invoked by, e.g.:

```
./smtrat path_to_your_smt_file.smt2
```

3 System architecture

SMT-RAT is a C++ library consisting of a collection of SMT compliant implementations of methods for solving non-linear real and integer arithmetic NRA/NIA formulas, we refer to as modules. These modules can be combined to (1) a theory solver in order to extend the supported logics of an existing SMT solver by NRA/NIA (see Figure 3.2) or (2) an SMT solver for NRA/NIA (see Figure 3.1). The latter is especially intended to be a testing environment for the development of SMT compliant implementations of further methods tackling NRA/NIA. Here, the developer only needs to implement the given interfaces of an SMT-RAT module and does not need to care about parsing input files, transforming formulas to conjunctive normal form or embedding a SAT solver in order to solve the Boolean skeleton of the given formula. Instead, SMT-RAT provides this and more features, such as lemma exchange, which will be explained in following (taken from the system architecture description of our SAT'15 submission).

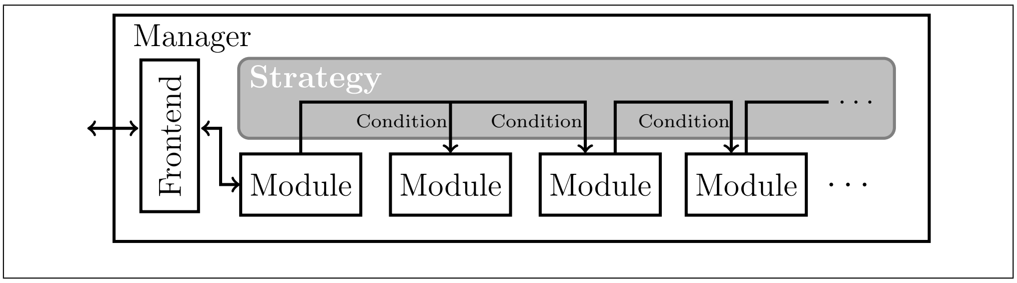


Figure 3.1: A snapshot of an SMT-RAT composition being an SMT solver for NRA.

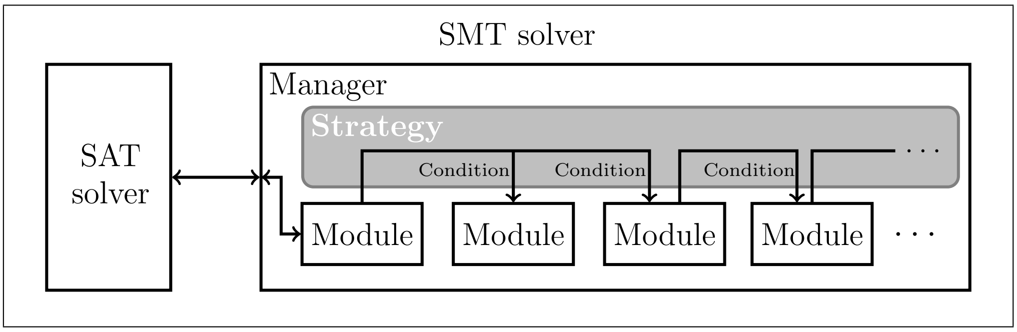


Figure 3.2: A snapshot of an SMT-RAT composition being a theory solver embedded in an SMT solver.

A *module* fixes a common interface for the SMT compliant implementations of procedures to tackle the satisfiability question of the supported logics of `SMT-RAT`. They can then be composed to a solver according to a user defined *strategy*. A *manager* maintains their allocation to *solving tasks* according to the strategy and provides the API, including the parsing of an `.smt2`-file.

3.1 Modules

A module m holds a set of formulas, called its *set of received formulas* and denoted by $C_{rcv}(m)$. The main function of a module is `check(bool full)`, which either decides whether $C_{rcv}(m)$ is satisfiable or not, returning `sat` or `unsat`, respectively, or returns `unknown`. A set of formulas is semantically defined by their conjunction. If the function's argument `full` is set to `false`, the underlying procedure of m is allowed to omit hard obstacles during solving at the cost of returning `unknown` in more cases. We can manipulate $C_{rcv}(m)$ by adding (removing) formulas φ to (from) it with `add(`φ`)` (`remove(`φ`)`). Usually, $C_{rcv}(m)$ is only slightly changed between two consecutive `check` calls, hence, the solver's performance can be significantly improved if a module works incrementally and supports backtracking. In case m determines the unsatisfiability of $C_{rcv}(m)$, it has to compute at least one preferably small *infeasible subset* $C_{inf}(m) \subseteq C_{rcv}(m)$. Moreover, a module can specify *lemmas*, which are valid formulas. They encapsulate information which can be extracted from a module's internal state and propagated among other modules. Furthermore, a module itself can ask other modules for the satisfiability of its *set of passed formulas* denoted by $C_{pas}(m)$, if it invokes the procedure `runBackends(bool full)` (controlled by the manager). It thereby delegates work to modules that may be more suitable for the problem at hand.

3.2 Strategy

`SMT-RAT` allows a user to decide how to compose the modules. For this purpose we provide a graphical user interface, where the user can create a *strategy* specifying this composition. A strategy is a directed tree $T := (V, E)$ with a set V of modules as nodes and $E \subseteq V \times \Omega \times \Sigma \times V$, with Ω being the set of *conditions* and Σ being the set of *priority values*. A condition is an arbitrary Boolean combination of formula properties, such as propositions about the Boolean structure of the formula, e.g., whether it is in conjunctive normal form (CNF), about the constraints, e. g., whether it contains equations, or about the polynomials, e.g., whether they are linear. Furthermore, each edge carries a unique priority value from $\Sigma = \{1, \ \ldots, \ |E|\}$.

3.3 Manager

The *manager* holds the strategy and the SMT solver's input formula C_{input}. Initially, the manager calls the method `check` of the module m_r given by the root of the strategy with $C_{rcv}(m_r) = C_{input}$. Whenever a module $m \in V$ calls `runBackends`, the manager adds a *solving task* $(\sigma, \ m, \ m')$ to its priority queue Q of solving tasks (ordered by the priority value), if there exists an edge $(m,\ \omega,\ \sigma,\ m') \in E$ in the strategy such that ω holds for $C_{pas}(m)$. If a processor p on the machine where `SMT-RAT` is executed on is available, the first solving task of Q is assigned to p and popped from Q. The manager thereby starts the method `check` of m' with $C_{rcv}(m') = C_{pas}(m)$ and passes the result (including infeasible subsets and lemmas) back to m. The module m can now benefit in its solving and reasoning process from this shared information. Note that a strategy-based composition of modules works incrementally and supports backtracking not just within one module but as a whole. This is realized by a mapping in each module m of its passed formulas $\varphi \in C_{pas}(m)$ to sets $R_1, \ldots,\ R_n \subseteq C_{rcv}(m)$ such that each R_i forms a reason why m included φ in $C_{pas}(m)$ to ask for its satisfiability. In order to exploit the incrementality of the modules, all parallel executed backends terminate in a consistent state (instead of just being killed), if one of them finds an answer.

3.4 Procedures implemented as modules

The heart of an SMT solver usually forms a SAT solver. In `SMT-RAT`, the module $\texttt{SAT}_M$ abstracts $C_{rcv}(\texttt{SAT}_M)$ to propositional logic and uses the efficient SAT solver `minisat` [5] to find a Boolean assignment of the abstraction. It invokes `runBackends` where $C_{pas}(\texttt{SAT}_M)$ contains the constraints abstracted by the assigned Boolean variables in a less-lazy fashion [11]. The module $\texttt{SIM}_M$ implements the Simplex method equipped with branch-and-bound and cutting-plane procedures as presented in [4]. We apply it on the linear constraints of any conjunction of NRA/NIA constraints. For a conjunction of nonlinear constraints `SMT-RAT` provides the modules $\texttt{GB}_M$, $\texttt{VS}_M$ and $\texttt{CAD}_M$, implementing GB [8], VS [2] and CAD [9] procedures, respectively. Moreover, the module $\texttt{ICP}_M$ uses ICP similar as presented in [6], lifting splitting decisions and contraction lemmas to a preceding $\texttt{SAT}_M$ and harnessing other modules for nonlinear conjunctions of constraints as backends. The exact procedure is going to be published. The module $\texttt{CNF}_M$ invokes `runBackends` on $C_{pas}(\texttt{CNF}_M)$ being the CNF of $C_{rcv}(\texttt{CNF}_M)$, and the module $\texttt{PP}_M$ performs some preprocessing based on factorizations and sum-of-square decompositions of polynomials.

3.5 Infeasible subsets and lemmas

Infeasible subsets and lemmas, which contain only formulas from $C_{pas}(\texttt{M}_{\texttt{SAT}})$ of a preceding $\texttt{M}_{\texttt{SAT}}$, prune its Boolean search space and hence the number of theory calls. Smaller infeasible subsets are usually more advantageous, because they make larger cuts in the search space. We call lemmas containing new constraints *inventive lemmas* (*non-inventive* otherwise). They might enlarge the Boolean search space, but they can reduce the complexity of later theory calls. When using inventive lemmas, it is important to ensure that the set possible constraints introduced in such lemmas is finite for a given module and a given input formula. Otherwise, the termination of this procedure cannot be guaranteed. In general, any module might contribute lemmas and all preceding modules in the solving hierarchy can directly involve them in their search for satisfiability.

4 Constructing an formula

The class `Formula` represents SMT formulas, which are defined according to the following abstract grammar

$$
\begin{array}{llccccccccccc}
p & ::= & a & | & b & | & x & | & (p+p) & | & (p \cdot p) & | & (p^e) \\
v & ::= & u & | & x & & & & & & & & \\
s & ::= & f(v,\ldots,v) & | & u & | & x & & & & & & \\
e & ::= & s = s & & & & & & & & & & \\
c & ::= & p = 0 & | & p < 0 & | & p \leq 0 & | & p > 0 & | & p \geq 0 & | & p \neq 0 \\
\varphi & ::= & c & | & (\neg\varphi) & | & (\varphi \land \varphi) & | & (\varphi \lor \varphi) & | & (\varphi \rightarrow \varphi) & | & \\
 & & (\varphi \leftrightarrow \varphi) & | & (\varphi \oplus \varphi) & & & & & & & &
\end{array}
$$

where a is a rational number, e is a natural number greater one, b is a *Boolean variable* and the *arithmetic variable* x is an inherently existential quantified and either real- or integer-valued. We call p a *polynomial* and use a `CArL` multivariate polynomial with `cln` rationals as coefficients to represent it. The *uninterpreted function* f is of a certain *order* $o(f)$ and each of its $o(f)$ arguments are either an arithmetic variable or an *uninterpreted variable* u, which is also inherently existential quantified, but has no domain specified. Than an *uninterpreted equation* e has either an uninterpreted function, an uninterpreted variable or an arithmetic variable as left-hand respectively right-hand side. A *constraint* c compares a polynomial to zero, using a *relation symbol*. Furthermore, we keep constraints in a normalized representation to be able to differ them better.

4.1 Normalized constraints

A normalized constraint has the form

$$
a_1 \, \overbrace{x_{1,1}^{e_{1,1}} \cdot \ldots \cdot x_{1,k_1}^{e_{1,k_1}}}^{m_1} + \ldots + a_n \, \overbrace{x_{n,1}^{e_{n,1}} \cdot \ldots \cdot x_{n,k_n}^{e_{n,k_n}}}^{m_n} \; + \; d \;\sim\; 0
$$

with $n \geq 0$, the *ith coefficient* a_i being an integral number ($\neq 0$), d being a integral number, x_{i,j_i} being a real- or integer-valued variable and e_{i,j_i} being a natural number greater zero (for all $1 \leq i \leq n$ and $1 \leq j_i \leq k_i$). Furthermore, it holds that $x_{i,j_i} \neq x_{i,l_i}$ if $j_i \neq l_i$ (for all $1 \leq i \leq n$ and $1 \leq j_i, l_i \leq k_i$) and $m_{i_1} \neq m_{i_2}$ if $i_1 \neq i_2$ (for all $1 \leq i_1, i_2 \leq n$). If n is 0 then d is 0 and $\sim$ is either $=$ or $<$. In the former case we have the normalized representation of any variable-free consistent constraint, which semantically equals `true`, and in the latter case we have the normalized representation of any variable-free inconsistent constraint, which semantically equals `false`. Note that the monomials and the variables in them are ordered according the graded lexicographic order of `CArL`. Moreover, the first coefficient of a normalized constraint (with respect to this order) is always positive and the greatest common divisor of $a_1, \ldots,\ a_n,\ d$ is 1. If all variable are integer valued the constraint is further simplified to

$$\frac{a_1}{g} \cdot m_1 \ + \ \ldots \ + \ \frac{a_n}{g} \cdot m_n \ + \ d' \ \sim' \ 0,$$

where g is the greatest common divisor of $a_1, \ldots,\ a_n$,

$$\sim' = \begin{cases} \leq, & \text{if } \sim \text{ is } < \\ \geq, & \text{if } \sim \text{ is } > \\ \sim, & \text{otherwise} \end{cases}$$

and

$$d' = \begin{cases} \lceil \frac{d}{g} \rceil & \text{if } \sim' \text{ is } \leq \\ \lfloor \frac{d}{g} \rfloor & \text{if } \sim' \text{ is } \geq \\ \frac{d}{g} & \text{otherwise} \end{cases}$$

If additionally $\frac{d}{g}$ is not integral and $\sim'$ is $=$, the constraint is simplified $0 < 0$, or if $\sim'$ is $\neq$, the constraint is simplified $0 = 0$.

We do some further simplifactions, such as the elimination of multiple roots of the left-hand sides in equations and inequalities with the relation symbol $\neq$, e.g., $x^3 = 0$ is simplified to $x = 0$. We also simplify constraints whose left-hand sides are obviously positive (semi)/negative (semi) definite, e.g., $x^2 \leq 0$ is simplified to $x^2 = 0$, which again can be simplified to $x = 0$ according to the first simplification rule.

4.2 Boolean combinations of constraints and Boolean variables

A formula is stored as a directed acyclic graph, where the intermediate nodes represent the Boolean operations on the sub-formulas represented by the successors of this node. The leaves (nodes without successor) contain either a Boolean variable, a constraint or an uninterpreted equality. Equal formulas, that is formulas being leaves and containing the same element or formulas representing the same operation on the same sub-formulas, are stored only once.

The construction of formulas, which are represented by the `Formula`, is mainly based on the presented abstract grammar. A formula being a leaf wraps the corresponding objects representing a Boolean variable, a constraint or an uninterpreted equality. A Boolean combination of Boolean variables, constraints and uninterpreted equalities consists of a Boolean operator and the sub-formulas it interconnects. For this purpose we either firstly create a set of formulas containing all sub-formulas and then construct the Formula or (if the formula shall not have more than three sub-formulas) construct the formula directly passing the operator and sub-formulas. Formulas, constraints and uninterpreted equalities are non-mutable, once they are constructed.

We give a small example constructing the formula

$$(\neg b \ \wedge \ x^2 - y < 0 \ \wedge \ 4x + y - 8y^7 = 0) \ \rightarrow \ (\neg(x^2 - y < 0) \ \vee \ b),$$

with the Boolean variable b and the real-valued variables x and y, for demonstration. Furthermore, we construct the UF formula

$$v = f(u, u) \ \oplus \ w \neq u$$

with u, v and w being uninterpreted variables of not specified domains S and T, respectively, and f is an uninterpreted function with not specified domain $T^{S \times S}$.

Firstly, we show how to create real valued (integer valued analogously with `carl::VariableType::VT_INT`), Boolean and uninterpreted variables:

```
carl::Variable x = smtrat::newVariable( "x", carl::VariableType::VT_REAL );
carl::Variable y = smtrat::newVariable( "y", carl::VariableType::VT_REAL );
carl::Variable b = smtrat::newVariable( "b", carl::VariableType::VT_BOOL );
carl::Variable u = smtrat::newVariable( "u", carl::VariableType::VT_UNINTERPRETED );
carl::Variable v = smtrat::newVariable( "v", carl::VariableType::VT_UNINTERPRETED );
carl::Variable w = smtrat::newVariable( "w", carl::VariableType::VT_UNINTERPRETED );
```

Uninterpreted variables, functions and function instances combined in equations or inequalities comparing them are constructed the following way.

```
carl::Sort sortS = smtrat::newSort( "S" );
carl::Sort sortT = smtrat::newSort( "T" );
carl::UVariable uu( u, sortS );
carl::UVariable uv( v, sortT );
carl::UVariable uw( w, sortS );
carl::UninterpretedFunction f = smtrat::newUF( "f", sortS, sortS, sortT );
carl::UFInstance f1 = smtrat::newUFInstance( f, uu, uw );
carl::UEquality ueqA( uv, f1, false );
carl::UEquality ueqB( uw, uu, true );
```

Next we see an example how to create polynomials, which form the left-hand sides of the constraints:

```
smtrat::Poly px( x );
smtrat::Poly py( y );
smtrat::Poly lhsA = px.pow(2) - py;
smtrat::Poly lhsB = smtrat::Rational(4) * px + py - smtrat::Rational(8) * py.pow(7);
```

Constraints can then be constructed as follows:

```
smtrat::ConstraintT constraintA( lhsA, carl::Relation::LESS );
smtrat::ConstraintT constraintB( lhsB, carl::Relation::EQ );
```

Now, we can construct the atoms of the Boolean formula

```
smtrat::FormulaT atomA( constraintA );
smtrat::FormulaT atomB( constraintB );
smtrat::FormulaT atomC( b );
smtrat::FormulaT atomD( ueqA );
smtrat::FormulaT atomE( ueqB );
```

and the formulas itself (either with a set of arguments or directly):

```
smtrat::FormulasT subformulasA;
subformulasA.insert( smtrat::FormulaT( carl::FormulaType::NOT, atomC ) );
subformulasA.insert( atomA );
subformulasA.insert( atomB );
smtrat::FormulaT phiA( carl::FormulaType::AND, std::move(subformulasA) );
smtrat::FormulaT phiB( carl::FormulaType::NOT, atomA )
smtrat::FormulaT phiC( carl::FormulaType::OR, phiB, atomC );
smtrat::FormulaT phiD( carl::FormulaType::IMPLIES, phiA, phiC );
smtrat::FormulaT phiE( carl::FormulaType::XOR, atomD, atomE );
```

Note, that $\land$ and $\lor$ are n-ary constructors, $\neg$ is a unary constructor and all the other Boolean operators are binary.

5 Embedding of an SMT-RAT solver composition

In this chapter we show how to embed a solver composed as explained in Chapter 7, e. g., using the default strategy solver `RatOne`. For instance, we could embed a theory solver composed with `SMT-RAT` into an SMT solver in order to extend its supported logics by NRA/NIA (or any of its sub-logics) or embed an SMT solver composed with `SMT-RAT` into a model checker for the verification of the satisfiability/unsatisfiability of occurring SMT formulas.

If for instance the SMT solver based on the strategy of `RatOne` shall be used (we can also choose any self-composed strategy here), we can create it as follows:

```
smtrat::RatOne yourSolver = smtrat::RatOne();
```

In Chapter 4 we have seen, how to construct an object representing an SMT formula. Having this formula, we can add it to the formulas, whose conjunction the solver composed with `SMT-RAT` has to check later for satisfiability. Here we give an overview of all interfaces:

- `bool inform( const FormulaT& )`

  Informs the solver about a constraint, wrapped by the given formula. Optimally, the solver should be informed about all constraints, which it will receive eventually, before any of them is added as part of a formula with the interface `add(..)`. The method returns `false` if it is easy to decide (for any module used in this solver), whether the constraint itself is inconsistent.

- `bool add( const FormulaT& )`

  Adds the given formula to the conjunction of formulas, which will be considered for the next satisfiability check. The method returns `false`, if it is easy to decide whether the just added formula is not satisfiable in the context of the already added formulas. Note, that only a very superficial and cheap satisfiability check is performed and mainly depends on solutions of previous consistency checks. In the most cases this method returns `true`, but in the case it does not the corresponding infeasible subset(s) can be obtained by `infeasibleSubsets()`.

- `Answer check( bool )`

  This method checks the so far added formulas for satisfiability. If, for instance we extend an SMT solver by a theory solver composed with `SMT-RAT`, these formulas are only constraints. The answer can either be `True`, if satisfiability has been detected, or `False`, if the formulas are not satisfiable, and `Unknown`, if the composition cannot give a conclusive answer. If the answer has been `True`, we get the model, satisfying the conjunction of the given formulas, using `model()` and, if it has been `False`, we can obtain infeasible subsets by `infeasibleSubsets()`. If the answer is `Unknown`, the composed solver is either incomplete (which highly depends on the strategy but for NRA it is actually always possible to define a strategy for a complete `SMT-RAT` solver) or it communicates lemmas/tautologies, which can be obtained applying `lemmas()`. If we embed, e.g., a theory solver composed with `SMT-RAT` into an SMT solver, these lemmas can be used in its SAT solving process in the same way as infeasible subsets are used. The strategy of an SMT solver composed with `SMT-RAT` has to involve a $\texttt{SAT}_M$ before any theory module is used[1] and, therefore, the SMT solver never communicates these lemmas as they are already processed by the $\texttt{SAT}_M$. A better explanation on the modules and the strategy can be found in Chapter 3 and Chapter 7. If the Boolean argument of the function `check` is `false`, the composed solver is allowed to omit hard obstacles during solving at the cost of returning `unknown` in more cases.

- `void push()`

  Pushes a backtrack point to the stack of backtrack points.

- `bool pop()`

  Pops a backtrack point from the stack of backtrack points and undoes everything which has been done after adding that backtrack point. It returns `false` if no backtrack point is on the stack. Note, that `SMT-RAT` supports incrementality, that means, that by removing everything which has been done after adding a backtrack point, we mean, that all intermediate solving results which only depend on the formulas to remove are deleted. It is highly recommended not to remove anything, which is going to be added directly afterwards.

- `const std::vector<FormulasT>& infeasibleSubsets() const`

  Returns one or more reasons for the unsatisfiability of the considered conjunction of formulas of this `SMT-RAT` composition. A reason is an infeasible subset of the sub-formulas of this conjunction.

- `const Model model() const`

  Returns an assignment of the variables, which occur in the so far added formulas, to values of their domains, such that it satisfies the conjunction of these formulas. Note, that an assignment is only provided if the conjunction of so far added formulas is satisfiable. Furthermore, when solving non-linear real arithmetic formulas the assignment could contain other variables or freshly introduced variables.

[1] It is possible to define a strategy using conditions in a way, that we achieve an SMT solver, even if for some cases no $\texttt{SAT}_M$ is involved before a theory module is applied.

- `std::vector<FormulaT> lemmas() const`

  Returns valid formulas for the purposes as explained in Section 3.5. Note, that for instance the `ICP`$_M$ might return lemmas being splitting decisions, which need to be processed in, e. g., a SAT solver. A *splitting decision* has in general the form

  $$(c_1 \wedge \ldots \wedge c_n) \ \rightarrow (p \leq r \ \vee \ p > r)$$

  where $c_1, \ldots, \ c_n$ are constraints of the set of currently being checked constraints (forming a *premise*), p is a polynomial (in the most cases consisting only of one variable) and $r \in \mathbb{Q}$. Hence, splitting decisions always form a tautology. We recommend to use the `ICP`$_M$ only in strategies with a preceding `SAT`$_M$. The same holds for the `SIM`$_M$, `VS`$_M$, and `CAD`$_M$ if used on NIA formulas. Here, again, splitting decisions might be communicated.

6 Implementing further modules

In this chapter we explain how to implement further modules. A module is a derivation of the class `Module` and we give an introduction to its members, interfaces and auxiliary methods in the following of this chapter. A new module and, hence, the corresponding `C++` source and header files can be easily created when using the script `writeModules.py`. Its single argument is the module's name and the script creates a new folder in `src/lib/modules/` containing the source and header file with the interfaces yet to implement. Furthermore, it is optional to create the module having a template parameter forming a settings object as explained in Section 6.4. A new module should be created only this way, as the script takes care of a correct integration of the corresponding code into `SMT-RAT`. A module can be deleted belatedly by just removing the complete folder it is implemented in.

6.1 Main members of a module

Here is an overview of the most important members of the class `Module`.

- `vector<FormulasT> mInfeasibleSubsets`

  Stores the infeasible subsets of the so far received formulas, if the module determined that their conjunction is not satisfiable.

- `Manager* const mpManager`

  A pointer to the manager which maintains the allocation of modules (including this one) to other modules, when they call a backend for a certain formula. For further details see Chapter 7.

- `const ModuleInput* mpReceivedFormula`

  The *received formula* stores the conjunction of the so far received formulas, which this module considers for a satisfiability check. These formulas are of the type `Formula` and the `ModuleInput` is basically a list of such formulas, which never contains a formula more than once.

- `ModuleInput* mpPassedFormula`

  The *passed formula* stores the conjunction of the formulas which this module passes to a backend to be solved for satisfiability. There are dedicated methods to change this member, which are explained in the following.

The received formula of a module is the passed formula of the preceding module. The owner is the preceding module, hence, a module has only read access to its received formula. The `ModuleInput` also stores a mapping of a sub-formula in the passed formula of a module to its origins in the received formula of the same module. Why this mapping is essential and how we can construct it is explained in Section 6.3.

6.2 Interfaces to implement

In the following we explain which methods must be implemented in order to fill the module's interfaces with life. All these methods are the core implementation and wrapped by the actual interfaces. This way the developer of a new module needs only to take care about the implementation of the actual procedure for the satisfiability check. All infrastructure-related actions are performed by the actual interface.

6.2.1 Informing about a constraint

```
bool MyModule::informCore( const Formula& _constraint )
{
    // Write the implementation here.
}
```

Figure 6.1: Example showing how to implement the method `informCore`.

Informs the module about the existence of the given constraint (actually it is a formula wrapping a constraint) usually before it is actually added to this module for consideration of a later satisfiability check. At least it can be expected, that this method is called, before a formula containing the given constraint is added to this module for consideration of a later satisfiability check. This information might be useful for the module, e.g., for the initialization of the data structures it uses. If the module can already decide whether the given constraint is not satisfiable itself, it returns `false` otherwise `true`.

6.2.2 Adding a received formula

```
bool MyModule::addCore( const ModuleInput::const_iterator )
{
    // Write the implementation here.
}
```

Figure 6.2: Example showing how to implement the method `addCore`.

Adds the formula at the given position in the conjunction of received formulas, meaning that this module has to include this formula in the next satisfiability check. If the module can already decide (with very low effort) whether the given formula is not satisfiable in combination with the already received formulas, it returns `false` otherwise `true`. This is usually determined using the solving results this module has stored after the last consistency checks. In the most cases the implementation of a new module needs some initialization in this method.

6.2.3 Removing a received formula

```
void MyModule::removeCore( const ModuleInput::iterator )
{
    // Write the implementation here.
}
```

Figure 6.3: Example showing how to implement the method `removeCore`.

Removes the formula at the given position from the received formula. Everything, which has been stored in this module and depends on this formula must be removed.

6.2.4 Checking for satisfiability

```
Answer MyModule::checkCore( bool )
{
    // Write the implementation here.
}
```

Figure 6.4: Example showing how to implement the method `checkCore`.

Implements the actual satisfiability check of the conjunction of formulas, which are in the received formula. There are three options how this module can answer: it either determines that the received formula is satisfiable and returns `True`, it determines unsatisfiability and returns `False`, or it cannot give a conclusive answer and returns `Unknown`. A module has also the opportunity to reason about the conflicts occurred, if it determines unsatisfiability. For this purpose it has to store at least one infeasible subset of the set of so far received formulas. If the method `check` is called with its argument being `false`, this module is allowed to omit hard obstacles during solving at the cost of returning `unknown` in more cases, we refer to as a *lightweight check*.

6.2.5 Updating the model/satisfying assignment

If this method is called, the last result of a satisfiability check was `True` and no further formulas have been added to the received formula, this module needs to fill its member `mModel` with a model. This model must be complete, that is all variables and uninterpreted functions occurring in the received formula must be assigned to a value of their corresponding domain. It might be necessary to involve the backends using the method `getBackendsModel()` (if they have been asked for the satisfiability of a sub-problem). It stores the model of one backend into the model of this module.

```
void MyModule::updateModel()
{
    // Write the implementation here.
}
```

Figure 6.5: Example showing how to implement the method `updateModel`.

6.3 Running backend modules

Modules can always call a backend in order to check the satisfiability of any conjunction of formulas. Fortunately, there is no need to manage the assertion of formulas to or removing of formulas from the backend. This would be even more involved as we do allow changing the backend if it is appropriate (more details to this are explained in Chapter 7). Running the backend is done in two steps:

1. Change the passed formula to the formula which should be solved by the backend. Keep in mind, that the passed formula could still contain formulas of the previous backend call.

2. Call `runBackends( full )`, where `full` being `false` means that the backends have to perform a lightweight check.

The first step is a bit more tricky, as we need to know which received formulas led to a passed formula. For this purpose the `ModuleInput` maintains a mapping from a passed sub-formula to one or more conjunctions of received sub-formulas. We give a small example. Let us assume that a module has so far received the following constraints (wrapped in formulas)

$$c_0 : x \leq 0,\ c_1 : x \geq 0,\ c_2 : x = 0$$

and combines the first two constraints c_0 and c_1 to c_2. Afterwards it calls its backend on the only remaining constraint, that means the passed formula contains only $c_2 : x = 0$. The mapping of c_2 in the passed formula to the received sub-formulas it stems from then is

$$c_2 \ \mapsto\ (c_0 \wedge c_1,\ c_2).$$

The mapping is maintained automatically and offers two methods to add formulas to the passed formulas:

-
  ```
  std::pair<ModuleInput::iterator,bool>
  addReceivedSubformulaToPassedFormula
  (
      ModuleInput::const_iterator
  )
  ```

  Adds the formula at the given positition in the received formula to the passed formulas. The mapping to its *original formulas* contains only the set consisting of the formula at the given position in the received formula.

-
  ```
  std::pair<ModuleInput::iterator,bool> addSubformulaToPassedFormula
  (
      const Formula&
  )

  std::pair<ModuleInput::iterator,bool> addSubformulaToPassedFormula
  (
      const Formula&,
      const Formula&
  )

  std::pair<ModuleInput::iterator,bool> addSubformulaToPassedFormula
  (
      const Formula&,
      std::shared_ptr<std::vector<FormulaT>>&
  )
  ```

  Adds the given formula to the passed formulas. It is mapped to the given conjunctions of origins in the received formula. The second argument (if it exists) must only consist of formulas in the received formula. It returns a pair of a position in the passed formula and a `bool`. The `bool` is `true`, if the formula at the given position in the received formula has been added to the passed formula, which is only the case, if this formula was not yet part of the passed formula. Otherwise, the `bool` is `false`. The returned position in the passed formula points to the just added formula.

  The vector of conjunctions of origins can be passed as a shared pointer, which is due to a more efficient manipulation of these origins. Some of the current module implementations directly change this vector and thereby achieve directly a change in the origins of a passed formula.

If, by reason of a later removing of received formulas, there is no conjunction of original formulas of a passed formula left (empty conjunction are removed), this passed formula will be automatically removed from the backends and the passed formula. That does also mean, that if we add a formula to the passed formula without giving any origin (which is done by the first version of `addSubformulaToPassedFormula`), the next call of `removeSubformula` of this module removes this formula from the passed formula. Specifying received formulas being the origins of a passed formula highly improves the incremental solving performance, so we recommend to do so.

The second step is really just calling `runBackends` and processing its return value, which can be `True`, `False`, or `Unknown`.

6.4 Auxilliary functions

The `module` class provides a rich set of methods for the analysis of the implemented procedures in a module and debugging purposes. Besides all the printing methods, which print the contents of a member of this module to the given output stream, SMT-RAT helps to maintain the correctness of new modules during their development. It therefore provides methods to store formulas with their assumed satisfiability status in order to check them belatedly by any SMT solver which is capable to parse .smt2 files and solve the stored formula. To be able to use the following methods, the compiler flag SMTRAT_DEVOPTION_Validation must be activated, which can be easily achieved when using, e.g., ccmake.

- `static void addAssumptionToCheck( const X&, bool, const string& )`

  Adds the given formulas to those, which are going to be stored as an .smt2 file, with the assumption that they are satisfiable, if the given Boolean argument is true, or unsatisfiable, if the given Boolean argument is false. The formulas can be passed as one of the following types (X can be one of the following data structures)

  - Formula (a single formula of any type)
  - ModuleInput (the entire received or passed formula of a module)
  - FormulasT (a set of formulas, which is considered to be a conjunction)
  - ConstraintsT (a set of constraints, which is considered to be a conjunction)

  The third argument of this function is any string which helps to identify the assumption, e.g., involving the name of the module and for which purpose this assumption has been made.

- `static void storeAssumptionsToCheck( const Manager& )`

  This method stores all collected assumptions to the file assumptions.smt2, which can be checked later by any SMT solver which is capable to parse .smt2 files and solve the stored formula. As this method is static, we need to pass the module's manager (*mpManager). Note that this method will be automatically called when destructing the given manager. Invoking this method is only reasonable, if the solving aborts directly afterwards and, hence, omits the manager's destructor.

-
  ```
  void checkInfSubsetForMinimality
  (
      vector<FormulasT>::const_iterator,
      const string&,
      unsigned
  ) const
  ```

  This method checks the infeasible subset at the given position for minimality, that is it checks whether there is a subset of it having maximally n elements less while still being infeasible. As for some approaches it is computationally too hard to provide always a minimal infeasible subset, they rather provide infeasible subsets not necessarily being minimal. This method helps to analyze how close the size of the encountered infeasible subsets is to a minimal one.

- Another important feature during the development of a new module is the collection of statistics. The script `writeModules.py` for the creation of a new module automatically adds a class to maintain statistics in the same folder in which the module itself is located. The members of this class store the statistics usually represented by primitive data types as integers and floats. They can be extended as one pleases and be manipulated by methods, which have also to be implemented in this class. `SMT-RAT` collects and prints these statistics automatically, if its command line interface is called with the option `--statistics` or `-s`.

- If the script `writeModules.py` for the creation of a new module is called with the option `-s`, the module has also a template parameter being a settings object. The different settings objects are stored in the settings file again in the same folder as the module is located. Each of these setting objects assigns all settings, which are usually of type `bool`, to values. The name of these objects must be of the form XYSettingsN, if the module is called XYModule and with N being preferably a positive integer. Fulfilling these requirements, the settings to compile this module with, can be chosen, e. g., with `ccmake`, by setting the compiler flag `SMTRAT_XY_Settings` to N.

  Within the implementation of the module, its settings can then be accessed using its template parameter `Settings`. If, for instance, we want to change the control flow of the implemented procedure in the new module depending on a setting `mySetting` being `true`, we write the following:

```
..
if(Settings::mySettings)
{
    ..
}
..
```

  This methodology assures that the right control flow is chosen during compilation and, hence, before runtime.

7 Composing a solver

SMT-RAT contributes a toolbox for composing an SMT compliant solver for its supported logics, that means it is incremental, supports backtracking and provides reasons for inconsistency. The resulting solver is either a fully operative SMT solver, which can be applied directly on .smt2-files, or a theory solver, which can be embedded into an SMT solver in order to extend its supported logics by those provided by SMT-RAT.

We are talking about composition and toolbox, as SMT-RAT contains implementations of many different procedures to tackle, e. g., NRA/NIA, each of them embedded in a module with uniform interfaces. These modules form the tools in the toolbox and it is dedicated to a user how to use them for solving an SMT formula. We provide a self-explanatory graphical user interface (GUI) for the definition of a graph-based strategy specifying which module(s) should be applied on which formula, taking into account the modules which were already involved.

In Section 3.2 we have already introduced a strategy and in the following of this chapter we firstly give a brief introduction to the existing modules equipped with an estimation of their input-based performances and then illustrate how to use the GUI for composing a strategy.

7.1 Existing module implementation

7.1.1 The CNFerModule

Transforms its received formula into conjunctive normal form CNF.

Efficiency The worst case complexity of this module is polynomial in the number of operators in the formula to transform.

7.1.2 The SATModule

This module abstracts it's received formula, being any SMT formula of the supported logics of SMT-RAT, to it's Boolean skeleton. It thereby replaces all constraints in the formula by fresh Boolean variables. The resulting propositional formula is then solved with minisat [5], where after each completed decision level the constraints belonging to the assigned Boolean variables are checked for consistency by the backends of this module. In the case of inconsistency, the infeasible subsets of the backends are abstracted and then involved in the search for a satisfying solution.

Efficiency Even though the worst case complexity of this procedure, not considering the complexity of the backend calls, is exponential in the number of variables in the abstracted formula, the procedure is in practice more efficient than any of the theory modules. Hence, it does clearly not form a bottleneck of the SMT solving. However, one should aim at reducing the number and complexity of the theory (backend) calls of this module, which might be influenced by infeasible subsets, which are small and/or involve constraints of earlier decision levels in the SAT solving, and lemmas, which either prune the search space of the SAT solving or ease subsequent theory calls.

7.1.3 The `LRAModule`

Implements the SMT compliant *Simplex* method presented in [4]. Hence, this module can decide the consistency of any conjunction consisting only of linear real arithmetic constraints. Furthermore, it might also find the consistency of a conjunction of constraints even if they are not all linear and calls a backend after removing some redundant linear constraints, if the linear constraints are satisfiable and the found solution does not satisfy the non-linear constraints. Note that the `LRAModule` might need to communicate a lemma/tautology to a preceding `SATModule`, if it receives a constraint with the relation symbol $\neq$ and the strategy needs for this reason to define a `SATModule` at any position before an `LRAModule`.

Integer arithmetic In order to find integer solutions, this module applies, depending on which settings are used, branch-and-bound, the construction of Gomory cuts and the generation of cuts from proofs [3]. It is also supported to combine these approaches. Note that for all of them the `LRAModule` needs to communicate a lemma/tautology to a preceding `SATModule` and the strategy needs for this reason to define a `SATModule` at any position before an `LRAModule`.

Efficiency The worst case complexity of the implemented approach is exponential in the number of variables occurring in the problem to solve. However, in practice, it performs much faster, and the worst case applies only on very artificial examples. This module outperforms any module implementing a method that is designed for solving formulas with non-linear constraints. If the received formula contains integer valued variables, the aforementioned methods might not terminate.

7.1.4 The `ICPModule`

Implements a combination of interval constraint propagation equipped with a Newton-based contraction [7] and LRA solving, for which we use our `LRAModule`. The implementation is inspired by [6], but additionally interacts with backends in order to exploit their efficiency on examples, where ICP fails. It thereby incorporates the possibility to invoke lightweight checks and highly benefits from the backends being optimized for small domains as, e. g., described in [9]. This module tries to lift splitting decisions as well as lemmas encoding a nonzero contraction to a preceding `SATModule`. It ensures an efficient processing of these decisions, which are this way shared with other modules.

Efficiency It is very difficult to give a conclusive statement about the efficiency of ICP. Usually, it performs better, if the domains of all variables are bounded intervals, preferably with a small diameter. It might also benefit from a higher number of constraints, as this introduces more chances for the propagation. However, more constraints mean also more overhead.

7.1.5 The `GroebnerModule`

Implements the Gröbner bases based procedure as presented in [8]. In general, this procedure can detect only the unsatisfiability of a conjunction of equations. This module also supports the usage of these equations to further simplify all constraints in the conjunction of constraints forming its input and passes these simplified constraints to its backends. However, it cannot be guaranteed that backends perform better on the simplified constraints than on the constraints before simplification.

Efficiency The worst case complexity of the underlying procedure is exponential in the number of variables of the input constraints. In the case that the conjunction of constraints to check for satisfiability contains equations, this module can be more efficient than other modules for NRA on finding out inconsistency.

7.1.6 The `VSModule`

Implements the *virtual substitution* method for a conjunction of constraints as described in [2]. This module supports incremental calls, efficient backtracking and infeasible subset generation. Note, that the infeasible subsets are often very small but not necessarily minimal. The implemented approach is not complete, as it maybe cannot decide the satisfiability of a conjunction containing a constraint, which involves a variable with degree 3 or more. Note, that even if no constraint of such form occurs in the received formula, this module might not be able to determine the consistency of its received formula, as it could create constraints of this form in its solving process. Nevertheless, the implemented approach is efficient compared to other approaches for non-linear real arithmetic conjunctions, and therefore well-suited to be used for solving conjunctions of non-linear real arithmetic constraints before complete approaches have their try. In combination with a backend, this module tries to solve the given problem and calls the backend on problems with less variables.

Efficiency The worst case complexity of this approach is exponential in the number of real arithmetic variables occurring in the conjunction to solve. It performs especially good on almost linear instances and slightly prefers problems only containing constraints with the relation symbols $\leq$, $\geq$ and $=$. It is often the case, that even if the conjunction to solve contains many not suited constraints, this module can determine the consistency on the basis of a well suited subset of the constraints in this conjunction.

7.1.7 The `CADModule`

This module implements an adapted version of the *cylindrical algebraic decomposition* (CAD) for a conjunction of constraints as described in [1]. It extends the original algorithm to be SMT compliant and implements the ideas from [9].

The CAD method consists of two basic routines: the projection (or elimination) of polynomials and the lifting (or construction) of samples. The projection transforms a set of polynomials over a set of variables to a new set of polynomials that do not contain some of the variables. The lifting starts with a sample point of degree k and constructs a sample point of degree $k+1$ using the polynomial sets from the projection. Both routines work in an incremental fashion: polynomials are only projected if needed and the construction is performed as a depth-first search.

Efficiency The worst case complexity of this algorithm is doubly exponential in the number of variables, the base being the sum of the number of polynomials and the maximum degree of any polynomial. This is due to an quadratic increase of polynomials in each projection step and a number of possible sample points that grows with the number of polynomials.

The practical performance heavily depends on the number and degree of polynomials created during the elimination. It benefits greatly if the real roots of the polynomials are rational, as irrational root operation may take quite some time.

7.2 Specifying a strategy with the GUI

The following subsections are used to give an overview of the SMT-RAT's GUI, which we call SMT-XRAT, and to introduce its functionalities.

7.2.1 Concept

The underlying concept of SMT-XRAT is the user-guided, visual modeling of module compositions in form of graphs and their mapping onto their corresponding source code for SMT-RAT. A modeled graph expresses an intended strategy graph of the user. Both can easily be projected on each other, because the data structure of a strategy graph also describes a graph structure, as explained in the previous chapter. A mapping considers not only the modeled hierarchy of the SMT-RAT modules, but also their attributes. Furthermore the GUI complies the constraints of these attributes during the modeling process, for example priority values are required to be unique.

The GUI does not only support the visual creation of strategy graphs and their translation into source code, but also enables the user to integrate the translated source code into SMT-RAT or, if necessary, delete it subsequently. The conclusive work only involves a recompilation of SMT-RAT with the desired strategy graph instance to obtain a customized SMT solver.

7.2.2 Main window structure

The main window structure of the SMT-XRAT application can be seen in Figure 7.1. It principally consists only of one large pane, which is called *strategy graph pane*. This pane embodies the workspace of the user and visualizes the composition of SMT-RAT modules, which are currently modeled. Only a comparatively small area is occupied by a compact menu bar, which offers for instance the exportation of a strategy graph into SMT-RAT.

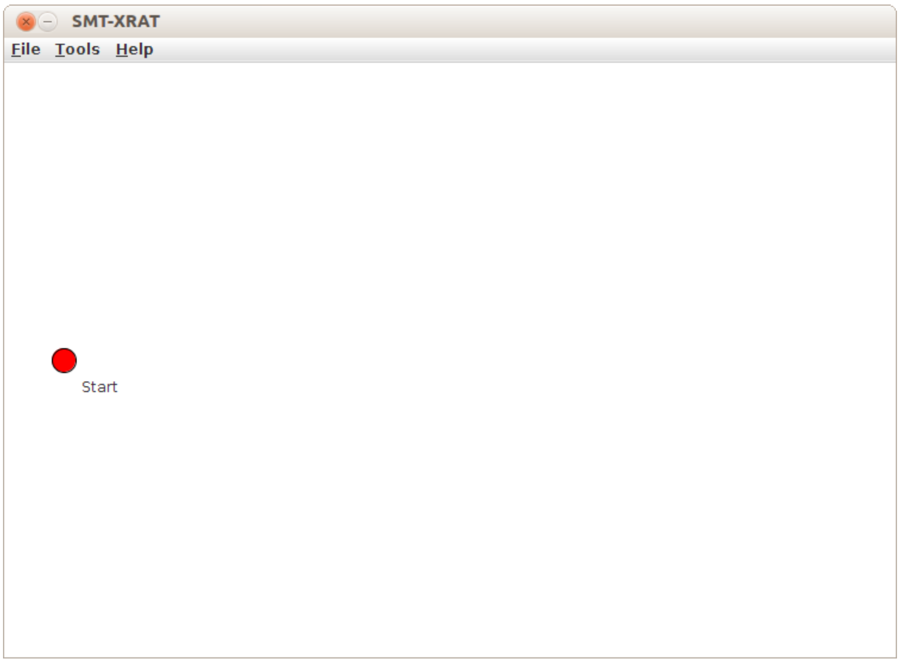


Figure 7.1: The main window of the SMT-XRAT application in its initial state.

7.2.3 Strategy graph pane

The graphs, which can be modeled in the strategy pane, must be acyclic, directed and weakly connected. Nodes represent SMT-RAT modules and edges represent the call hierarchy of them. Both of the element types are labeled to display all necessary and editable module attributes within the visualization. Modeling strategy graphs on the pane implies the interactive operations of adding, editing and deleting modules and also aligning elements, if desired by the user.

Adding backends

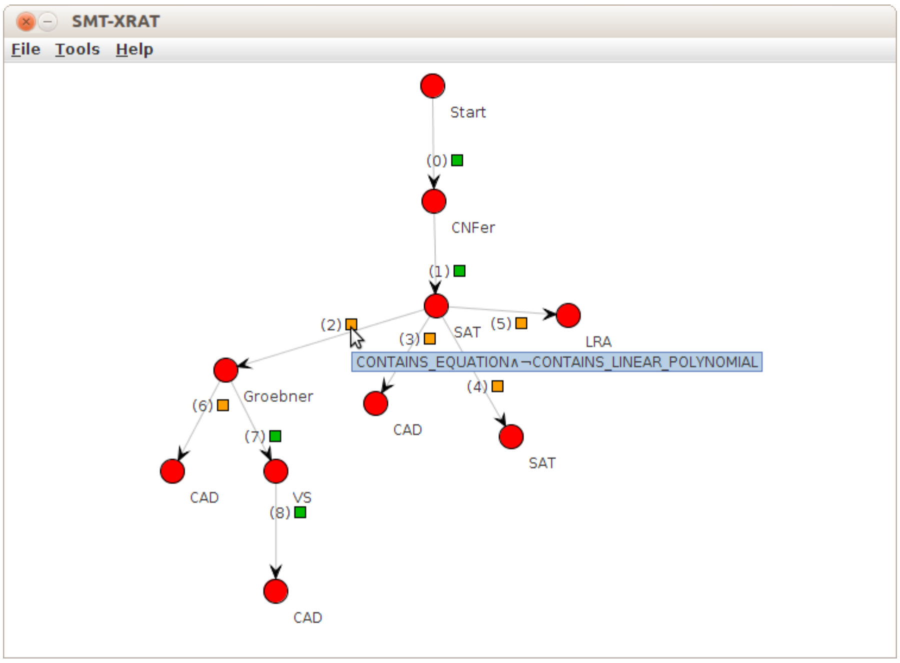


Figure 7.2: The small rectangles alongside the edges reveal the hidden condition of a backend.

An initial visualization of the pane contains the inevitable *Start* module of an SMT solver, which displays no attributes, but marks the starting point for the user to create the desired strategy graph. The user can simply consider the *Start* module as the front-end of an SMT solver where, e. g., NRA/NIA formulas are passed to. Building up a composition of modules occurs by appending backends to the *Start* module and then to the newly appended backends and so forth. When appending a backend to a selected module, a dialog window requests the operating user to input a condition and to choose the type of `SMT-RAT` module for the new backend. The GUI provides a special input interface to enter conditions, which is explained later. For each appended backend a new node as well as a directed edge from the originating module to this new node is drawn in the visualization. In this way, the graph gradually arises on the pane. A node is labeled with its type of `SMT-RAT` module whereas an edge holds its condition and an automatically assigned priority value. Initially this priority value is always the total number of currently existing modules minus 1, as the *Start* module is not counted. As the user-defined conditions might get quite long, they cannot be directly seen on the strategy graph pane. Instead, a small rectangle alongside the edge reveals them quickly on request. The user needs to point the mouse cursor over a rectangle to obtain its corresponding tool tip text, which shows the hidden condition, as can be seen in Figure 7.2. This leaves the graph compact and helps to concentrate on the more essential aspect of modeling an execution hierarchy. The user can choose to input an own condition or leave it by the default value of ‘`TRUE`’, which, as described before, means that given a formula the condition is always satisfied and, hence, the backends will always be used. To point out better which modules contain default conditions and which do not, the color of a rectangle containing a default condition is green and otherwise orange.

Grammar for conditions

When adding a module to the strategy graph pane, the user has to input a valid condition for its intended use as backend. A valid condition is a derivation of the formal Grammar

$$\mathcal{C} = (N, \Sigma, R, S).$$

The set of nonterminals is given by

$$N = \{S, T, B, C, D, P\},$$

whereas the set of terminal symbols

$$\Sigma = \{\texttt{(}, \texttt{)}, \neg, \leftrightarrow, \oplus, \rightarrow, \wedge, \vee\} \cup \{\texttt{TRUE}, \texttt{p}_1, \ldots, \texttt{p}_n\}$$

consists of the union of logical operators and propositions. $S \in N$ is the start symbol of the production rules denoted by the set R, which covers the following:

$$\begin{array}{lccccccccccc}
S & \rightarrow & \texttt{TRUE} & \mid & T & \mid & C & \mid & D & \mid & TBT \\
T & \rightarrow & P & \mid & \neg T & \mid & \texttt{(}C\texttt{)} & \mid & \texttt{(}D\texttt{)} & \mid & \texttt{(}TBT\texttt{)} \\
B & \rightarrow & \leftrightarrow & \mid & \oplus & \mid & \rightarrow & & & & \\
C & \rightarrow & C \wedge C & \mid & T & & & & & & \\
D & \rightarrow & D \vee D & \mid & T & & & & & & \\
P & \rightarrow & \texttt{p}_1 & \mid & \ldots & \mid & \texttt{p}_n & & & &
\end{array}$$

with the non-terminal symbols T being a term, B being a binary operator, C being a conjunction, D being a disjunction and P being a proposition.

The terminal symbols ‘$\neg$’, ‘$\leftrightarrow$’, ‘$\oplus$’, ‘$\rightarrow$’, ‘$\wedge$’ and ‘$\vee$’ represent their related logical operators, which, in the context of conditions, are negation, equivalence, exclusive or, implication, conjunction and disjunction respectively. Their semantics is defined as usual. The terminal symbols ‘`(`’ and ‘`)`’ are used, in case several different types of logical operators are utilized within one term. They point out the precedences of the operators in the same way as it is known from mathematical contexts. For example, for the term $p_1 \vee p_2 \wedge p_3$ it is unknown, which of the logical operators has the higher precedence. Writing the same term with parenthesis as $p_1 \vee \texttt{(}p_2 \wedge p_3\texttt{)}$ clarifies, that the conjunction operator is of higher precedence.

The propositions $P = \{p_1, \ldots, p_n\}$ are as explained in Section 3.2. This set can vary among releases of the `SMT-RAT`, as well as the user can also define own propositions. For this reason, the set of propositions is dynamically loaded from the `SMT-RAT` source code each time the GUI is started. As mentioned in the previous chapter, the set of SMT-RAT modules can vary as well. Therefore the list of available SMT-RAT modules is also dynamically loaded.

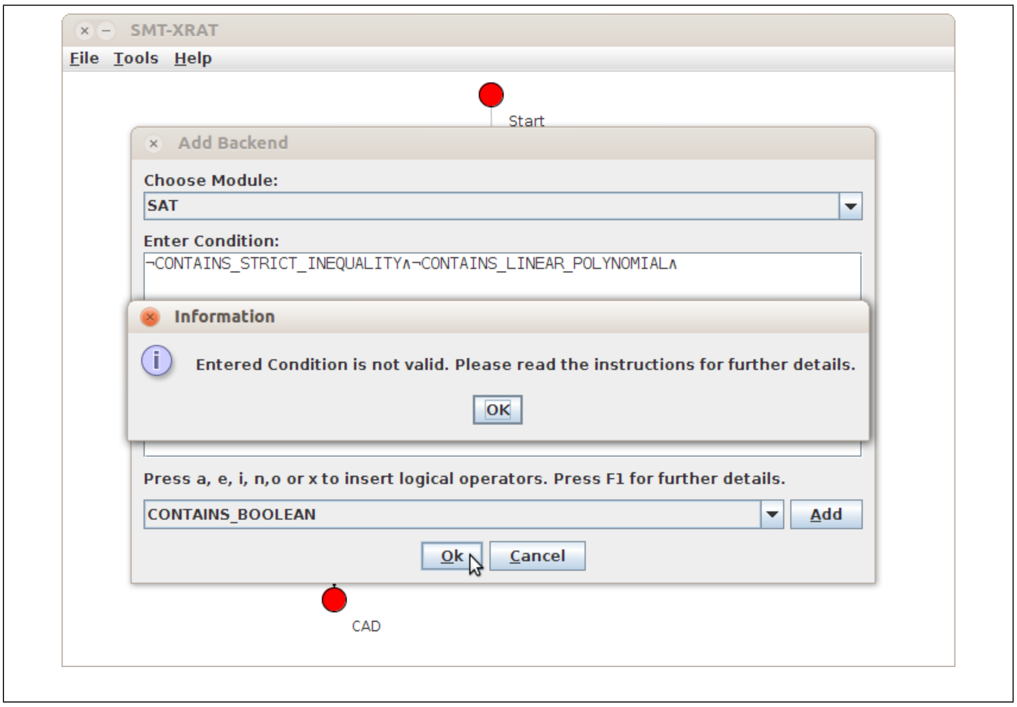
Interface for inputting conditions

When adding backends to existing modules on the strategy graph pane, a dialog window requests the user to input a condition, which must be derivable from the above defined Grammar $\mathcal{C}$. This dialog window is equipped with additional features to ease the input process for the user and to improve the usability. A specialized text area is used for inputting conditions. Initially, it contains the default proposition value 'TRUE'. The window also contains a combo box, where the user has the possibility to choose a proposition value from. A chosen proposition value can then be copied to the current caret position of the text area. Should the occasion arise that the user selects a part of an entered condition beforehand, it is simply overwritten by the chosen proposition value. The user can only input proposition values by using this combo box. Proposition values cannot be typed into the text area directly. On the one side, this simply prevents mistyping and, on the other side, the list of propositions might be changed between releases of SMT-RAT, as stated before. In many cases it will not be sufficient to use conditions, which contain just one single proposition. When requiring a Boolean combination of conditions, the above stated logical operators are needed. Although, the characters of the operators are generally not present on a keyboard, they can just be typed into the specialized text area of the dialog window. To type in the conjunction operator '$\wedge$', for instance, the user simply needs to hit the key 'c' on the keyboard. Instead of the character 'c', the character '$\wedge$' will then appear in the text area.

In order to increase the user experience even further, the text area treats the single characters of an inserted proposition value as a block, which cannot be entered by the caret of the text area. This means, that if the caret is positioned directly left of an inputted proposition value and the user navigates the caret to the right, it will jump to the position directly right of the proposition. The caret will never appear between the characters of a single proposition value. This is the analogous case for selecting and deleting proposition values. All characters of a proposition value are always selected, deselected or deleted at once.

The text area allows to copy and paste conditions or parts of it. Text, which should be pasted into the text area, is checked to guarantee, that it only contains allowed values. Otherwise it will be refused. Allowed values cover proposition values, the characters used to express logical operators and parenthesis.

When the user confirms the dialog window, the implemented recursive descent parser of SMT-XRAT checks, whether the inputted condition is a valid derivation of Grammar $\mathcal{C}$ or not. In case it is not, the user will be returned to the dialog window to re-edit the condition, as can be seen in Figure 7.3. Otherwise the inputted condition is adopted for the backend.



Choose Module: SAT

Enter Condition: ¬CONTAINS_STRICT_INEQUALITY∧¬CONTAINS_LINEAR_POLYNOMIAL∧

Entered Condition is not valid. Please read the instructions for further details.

Press a, e, i, n,o or x to insert logical operators. Press F1 for further details.

CONTAINS_BOOLEAN

Figure 7.3: A wrong condition has been inputted by the user.

Manipulating the strategy graph

Besides the capability of adding modules, the strategy graph pane gives the user also the possibility to remove and edit them subsequently.

The deletion of a single module implicates that all of its succeeding modules in the composition hierarchy will be removed as well. The strategy graph pane is only allowed to contain one weakly connected graph. Furthermore, when deleting one or implicitly more modules, the priority values of all remaining modules might automatically be adjusted to comply the constraints of the priority values (the priority value of an edge is always greater than the priority values of its preceding edges). However, the logical priority order remains untouched.

When editing modules, the same dialog window is displayed as for adding modules. The window components are already filled in with the attributes of the corresponding module. However, priority values are not manipulated via this dialog window. As said before, priority values are automatically assigned, when a module is created, and they are displayed alongside the edges. The user can manually change the priority order by pushing the priority value of a lower prioritized module in front of the priority value of a higher prioritized one. The user achieves this by using the mouse pointer to draw a dashed arrow from the edge label of that lower prioritized module to the edge label of the higher prioritized module, as it is illustrated by Figure 7.4. Afterwards the lower prioritized module will have a higher priority than the other one. The priority values of the modules might just be swapped. If this is not possible, the priority values of the modules and of their preceding modules are adjusted automatically, so that as a result, the newly prioritized module will be ordered logically before the other one. The adaptation of the priority values is emphasized by Figure 7.5.

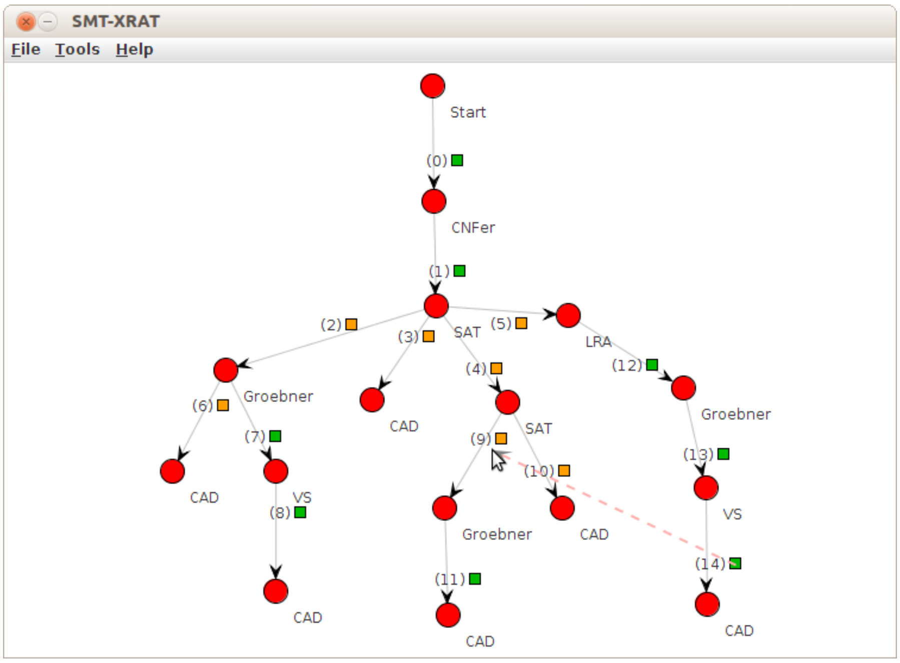


Figure 7.4: Priority values before changes are set.

7.2.4 Further functionalities

Further features of the GUI are reached through the menu bar. The most important and also necessary functionality is the management of strategy graphs inside the `SMT-RAT` source code, e. g., the translation of a strategy in the GUI into source code of `SMT-RAT`. To export a currently modeled strategy graph, the user simply needs to open the corresponding dialog window and choose a name. Figure 7.6 shows an example for exporting the current strategy graph and naming it `SMT-XRAT`. The GUI will then fulfill the translation and integration process. The same dialog window also lists all existing strategy graphs, which are currently integrated in the source code, and gives the opportunity to delete them separately. This can be seen for the existing strategy graph *NRATSolver* of the example.

The remaining features hold by the menu bar are not mandatory, but improve the creation process and usability. For example, the GUI allows the user to save the current strategy graph into an XML file. This file can then be opened again for later editing

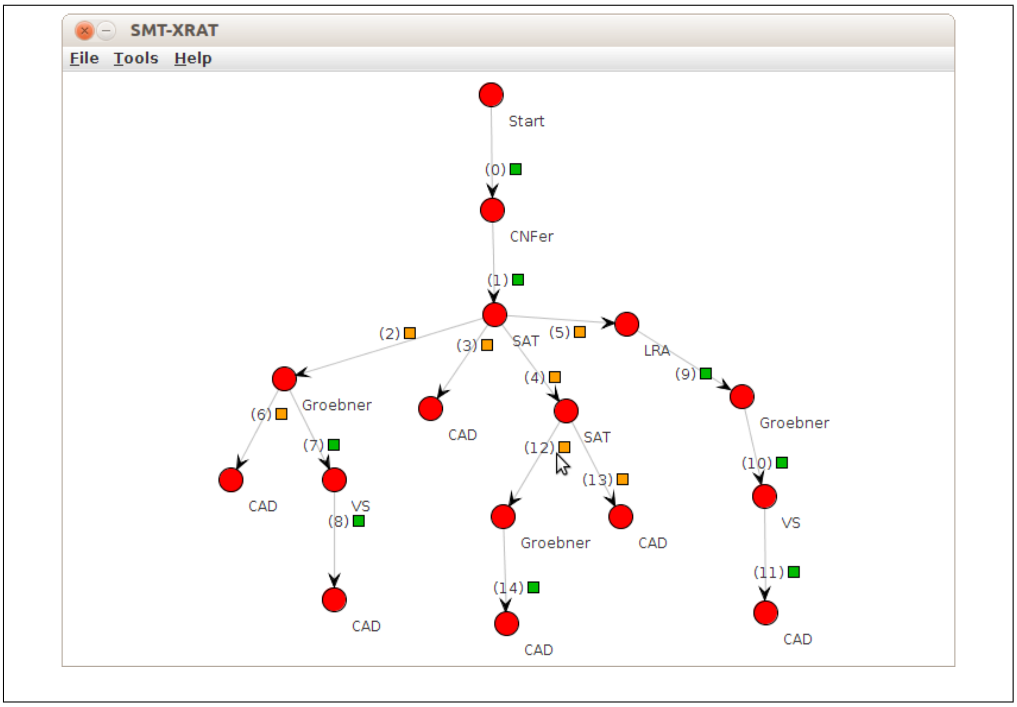
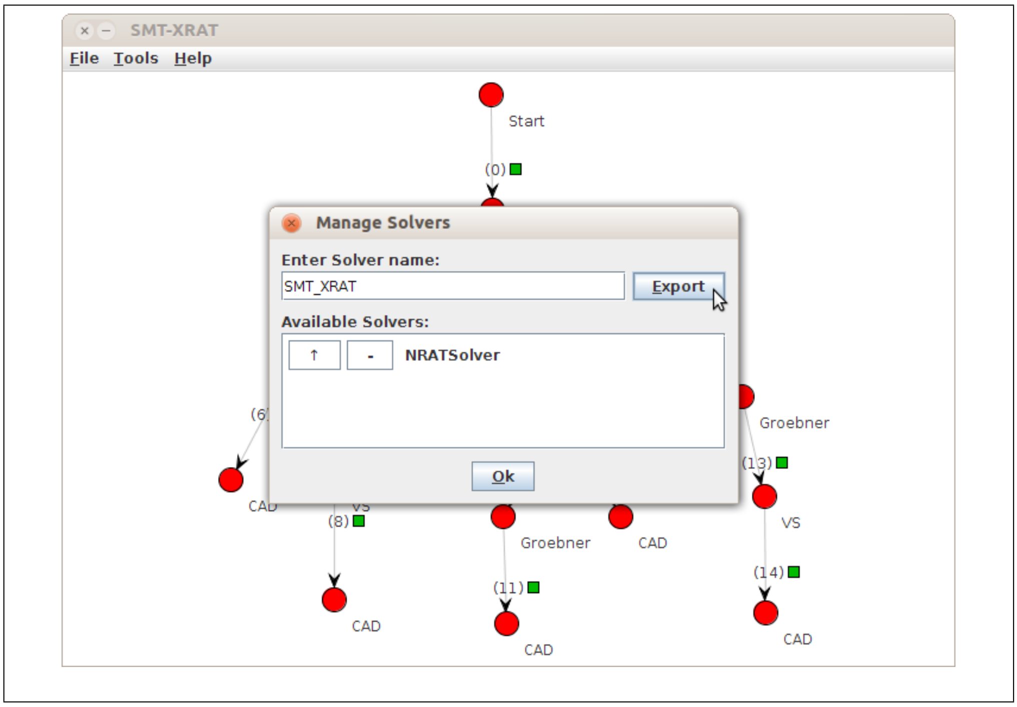


Figure 7.5: Intentionally changed and automatically adapted priority values.

or it can be exchanged with another user. Another practical feature is the ability to save a screen shot of the strategy graph pane into an image file. Such image files can be used to discuss strategy graphs, when it is not desired to run the GUI.



SMT-XRAT

Manage Solvers

Enter Solver name: SMT_XRAT

Available Solvers:

NRATSolver

Figure 7.6: Managing SMT solvers in the SMT-RAT source code.

8 Further features

8.1 Delta debugging

Delta debugging describes a generic debugging approach based on automated testing. Given a program and an input that provokes a certain behavior – for example an error – delta debugging is the process of iteratively changing the input, retaining the specific behavior. Each small change to the input represents a *delta* and is the result of some *transformation rule*. Whenever a change was successful, it is stored and the process continues from this intermediate result. Eventually, there is no transformation left, such that the faulty behavior is retained and the debugging process terminates.

This approach only works, if the transformation rules can neither be chained to form a loop, nor continue infinitely. Usually, as the ultimate goal is a minimal example that triggers some bug, all transformation rules are designed to make the input *smaller*, in one way or another.

This approach has proven to be very valuable in the context of SAT and SMT solving. However, existing delta debugging tools [10] needed a preprocessed input and manual restarts to achieve a fix-point, hence, we decided to include our own delta debugging tool, `delta`, in SMT-RAT. It can be used completely independent of SMT-RAT and is built to be as generic as possible, but focuses on programs operating on SMTLib files. It has some knowledge of the semantics of the corresponding logics, but only operates on nodes. Any SMTLib construct, that is either a constant or a braced expression, is a node.

The actual transformation rules are implemented in `operations.h` and are enabled in the constructor of the `Producer` class. The implemented rules are rather simple: removing a node, replacing a node by a child node, simplifying a number, replacing a symbol by a constant or eliminating a let expression. These transformations are designed such that they can be extended easily. For a given input `delta` applies each transformation to each node. Each application may produce arbitrarily many *candidate inputs* which are then tested. The first candidate that provokes the error is then adopted, the other candidates are rejected.

When analyzing the behavior, `delta` relies on the exit code of the program. It will run the program on the original input and obtain the *original exit code*. Whenever the program returns the same exit code, `delta` assumes that the program behaved the same. Hence, if you want to debug a specific assertion (or error, faulty output, ...), make sure that this event results in a unique exit code.

Using `delta` is rather easy. It accepts the input file and the solver as its two main arguments: `./delta -i input.smt2 -s ./solver`. There are a couple of other arguments that are documented in the help: `./delta --help`.

Bibliography

[1] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183. Springer, 1975.

[2] F. Corzilius and E. Ábrahám. Virtual substitution for SMT solving. In *Proc. of FCT'11*, volume 6914 of *LNCS*, pages 360–371. Springer, 2011.

[3] Isil Dillig, Thomas Dillig, and Alex Aiken. Cuts from proofs: a complete and practical technique for solving linear inequalities over integers. *Formal Methods in System Design*, 39(3):246–260, 2011.

[4] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[5] N. Eén and N. Sörensson. An extensible sat-solver. In *Proc. of SAT'03*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[6] S. Gao, M. K. Ganai, F. Ivancic, A. Gupta, S. Sankaranarayanan, and E. M. Clarke. Integrating ICP and LRA solvers for deciding nonlinear real arithmetic problems. In *Proc. of FMCAD'10*, pages 81–89. IEEE, 2010.

[7] Stefan Herbort and Dietmar Ratz. Improving the Efficiency of a Nonlinear-System-Solver Using a Componentwise Newton Method. 1997.

[8] S. Junges, U. Loup, F. Corzilius, and E. Ábrahám. On Gröbner bases in the context of satisfiability-modulo-theories solving over the real numbers. Technical Report AIB-2013-08, RWTH Aachen University, 2013.

[9] U. Loup, K. Scheibler, F. Corzilius, Erika E. Ábrahám, and Bernd Becker. A symbiosis of interval constraint propagation and cylindrical algebraic decomposition. In *Proc. of CADE-24*, volume 7898 of *LNCS*, pages 193–207. Springer, 2013.

[10] Aina Niemetz and Armin Biere. ddsmt: A delta debugger for the smt-lib v2 format. In *SMT Workshop 2013 11th International Workshop on Satisfiability Modulo Theories*, 2013.

[11] Roberto Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.