SMT-RAT

1 SMT-RAT 1

2 Installation 1

2.1 Requirements . . . 1
2.2 Download . . . 2
2.3 Configuration . . . 2
2.4 Build . . . 2
2.5 Check build . . . 2

3 System architecture 2

3.1 Different libraries . . . 2
3.2 Software design . . . 3
3.3 Modules . . . 4
3.4 Strategy . . . 4
3.5 Manager . . . 4
3.6 Procedures implemented as modules . . . 4
3.7 Infeasible subsets and lemmas . . . 4

4 Modules 5

4.1 Main members of a module . . . 5
4.2 Interfaces to implement . . . 5
4.2.1 Informing about a constraint . . . 5
4.2.2 Adding a received formula . . . 6
4.2.3 Removing a received formula . . . 6
4.2.4 Checking for satisfiability . . . 6
4.2.5 Updating the model/satisfying assignment . . . 6
4.3 Running backend modules . . . 7
4.4 Auxiliary functions . . . 8
4.5 Existing module implementation . . . 8
4.6 Available modules . . . 8
4.6.1 BEModule . . . 9
4.6.2 BVModule . . . 9
4.6.3 CNFerModule . . . 9
4.6.4 CSplitModule . . . 9
4.6.5 CoCoAGBModule . . . 9
4.6.6 CubeLIAModule . . . 9
4.6.7 CurryModule . . . 9
4.6.8 EMModule . . . 9
4.6.9 ESModule . . . 9
4.6.10 FPPModule . . . 9
4.6.11 FouMoModule . . . 9
4.6.12 GBModule . . . 9
4.6.13 GBPPModule . . . 10

4.6.14 ICEModule . . . 10
4.6.15 ICPModule . . . 10
4.6.16 IncWidthModule . . . 10
4.6.17 IntBlastModule . . . 10
4.6.18 IntEqModule . . . 10
4.6.19 LRAModule . . . 10
4.6.20 LVEModule . . . 11
4.6.21 MCBModule . . . 11
4.6.22 NRAILModule . . . 11
4.6.23 NewCADModule . . . 11
4.6.24 PBGaussModule . . . 11
4.6.25 PBPPModule . . . 11
4.6.26 PFEModule . . . 11
4.6.27 SATModule . . . 12
4.6.28 STropModule . . . 12
4.6.29 SplitSOSModule . . . 12
4.6.30 SymmetryModule . . . 12
4.6.31 UFCegarModule . . . 12
4.6.32 UnionFindModule . . . 12
4.6.33 VSModule . . . 12

5 Strategies **13**

6 Using SMT-RAT **14**
6.1 Standalone solver . . . 14
6.1.1 Formula analysis . . . 15
6.1.2 Preprocessing . . . 15
6.1.3 Quantifier elimination . . . 15
6.1.4 DIMACS solving . . . 15
6.1.5 Pseudo-Boolean solving . . . 15
6.1.6 Optimization . . . 16
6.2 Embedding in other software . . . 16
6.2.1 Interface . . . 16
6.2.2 Syntax of formulas . . . 17
6.2.3 Boolean combinations of constraints and Boolean variables . . . 17
6.2.4 Normalized constraints . . . 19
6.2.5 Linking . . . 19

7 Other tools **19**
7.1 Benchmarking . . . 19
7.2 Delta debugging . . . 19
7.2.1 SMT-RAT's own delta tool . . . 19
7.2.2 Using ddSMT . . . 20

7.3 Analyzer . . . 20
7.4 Preprocessing . . . 20
7.5 Benchmax . . . 20
7.5.1 General usage . . . 20
7.5.2 Tools . . . 21
7.5.3 Backends . . . 21
7.5.4 Troubleshooting . . . 22
7.5.5 Benchmax python utility . . . 22
7.6 Delta . . . 23
7.6.1 Delta debugging . . . 23

8 Developers information **23**
8.1 Code style . . . 24
8.2 Documentation . . . 24
8.2.1 Code comments . . . 24
8.2.2 Writing out-of-source documentation . . . 25
8.3 Settings . . . 25
8.4 Logging . . . 26
8.5 Statistics and timing . . . 26
8.6 Testing . . . 27
8.7 Validation . . . 27
8.8 Checkpoints . . . 28
8.9 Finding and Reporting Bugs . . . 29

9 NewCoveringModule #and **29**
9.1 Introduction . . . 29
9.2 Usage . . . 29
9.3 Efficiency . . . 30

1 SMT-RAT

This is the documentation of SMT-RAT, an Open Source C++ Toolbox for Strategic and Parallel SMT Solving. On this page, you can find introductory information on how to obtain and compile SMT-RAT and a traditional doxygen API documentation. The documentation comes in three flavours:

- API documentation as HTML: the regular web-based doxygen documentation is generated by `make doc-apidoc-html` into `build/doc/apidoc-html/` and online at https://smtrat.github.io/.
- API documentation as PDF: (almost) the full doxygen documentation as a pdf file is generated by `make doc-apidoc-pdf` into `build/doc/doc_smtrat-*.pdf` and online here.
- Manual as PDF: only the manual, suitable for reading as an introduction into SMT-RAT, is generated by `make doc-manual` into `build/doc/manual_smtrat-*.pdf` and online here.

Note that all the information of the manual is contained in the API documentation (both HTML and PDF) as well. It is much more compact, though, and may thus be more approachable as an introduction. However, references to classes do not work in the manual (as the class documentation is not contained).

If you are new to SMT-RAT and want to have a look around, we recommend reading the manual. The full API documentation can be found either online, or in the pdf documentation.

If you want to use SMT-RAT and want to know how to get and install it, have a look at Installation. It covers the most important steps including obtaining the actual source code, obtaining dependencies, building the library and running our test suite.

If you already use SMT-RAT and want to dig deeper or submit new code, you can additionally browse in Developers information. It contains information about supplementary features like our logging framework and some basic guidelines for our code like how we use doxygen.

Note that this documentation is, and will probably still be for quite some time, work in progress. If you feel that some topic that is important to you is missing or some explanation is unclear, please let us know!

2 Installation

2.1 Requirements

SMT-RAT is build and tested on Linux. While MacOS is a secondary target (and should work on the most recent version), we do not target Windows yet. Please contact us if you are interested in changing that.

To build and use SMT-RAT, you need the following other software:

- `git` to checkout the git repository.
- `g++` or `clang` to compile.
- `cmake` to generate the make files.
- `boost` for several additional libraries (automatically built locally if necessary).
- `gmp` for calculations with large numbers (automatically built locally if necessary).
- `carl` from `http://smtrat.github.io/carl/` (automatically built locally if necessary).

Optional dependencies

- `ccmake` to set cmake flags.
- `doxygen` to build the documentation.
- `ginac` to enable the usage of polynomial factorization.

When installing the dependencies, make sure that you meet the following version requirements:

- `g++` $>=$ 7
- `clang` $>=$ 5

2.2 Download

Here are archived versions of SMT-RAT for download:

- `latest`

We mirror our master branch to github.com. If you want to use the newest bleeding edge version, you can checkout from `https://github.com/smtrat/smtrat`. Although we try to keep the master branch stable, there is a chance that the current revision is broken. You can check `here` if the current revision compiles.

2.3 Configuration

SMT-RAT is configured with cmake. To prepare the build and perform the configuration run the following, starting from the root folder of SMT-RAT:

```
$ mkdir build && cd build && cmake ..
$ ccmake ..
```

`ccmake` will show the cmake variables. [TODO: document important cmake variables]

2.4 Build

To build SMT-RAT use `make` in the build folder:

```
$ make smtrat-shared
```

Relevant targets you may want to build individually include:

- `smtrat-shared`: Builds the (dynamically linked) executable SMT solver
- `smtrat-static`: Builds the (statically linked) executable SMT solver
- `smtrat`: Builds both `smtrat-shared` and `smtrat-static`.
- `doc-html`: Builds the doxygen documentation as HTML.
- `doc-pdf`: Builds the doxygen documentation as PDF.
- `doc`: Builds both `doc-html` and `doc-pdf`.
- `benchmax`: Builds the benchmarking tool.
- `delta`: Builds the delta debugger.

2.5 Check build

You can now find an executable `smtrat-shared` in the build directory. It shows some usage information if you run `./smtrat-shared --help`. To run it on an SMT-LIB file, simply run

```
$ ./smtrat-shared example.smt2
```

3 System architecture

3.1 Different libraries

THe different parts of SMT-RAT are split into multiple libraries (in the sense of a shared object library) that are responsible for the following tasks:

- `smtrat-analyzer`: static analysis of input formulae;
- `smtrat-cad`: back all CAD-based techniques;
- `smtrat-common`: common definitions and includes;
- `smtrat-max-smt`: takes care of max SMT queries;
- `smtrat-mcsat`: utilities for the MCSAT-based solver;
- `smtrat-modules`: all regular SMT-RAT modules;
- `smtrat-optimization`: takes care of optimization queries;
- `smtrat-qe`: methods for quantifier elimination;
- `smtrat-solver`: core solving infrastructure;
- `smtrat-strategies`: strategies for SMT solving;
- `smtrat-unsat-cores`: takes care of unsat core computations.

All of these yield a library, while a full-fledged SMT solver is build from the `cli/` path, in particular the `cli/smtratSolver.cpp`.

3.2 Software design

The architecture of SMT-RAT puts its focus on modularity and composability of different solving techniques. Every solving technique, for example SAT solving or the simplex method, is encapsulated in a (derivation of the) module class. These modules are composed to a strategy that governs which modules are used in what order. The execution of a strategy is incumbent upon the manager, that also offers an interface for basic SMT solving to the outside.

More advanced solving techniques like quantifier elimination, computing unsatisfiable cores, or tackling max SMT and optimization queries are implemented as individual components in the frontend. The frontend implements (most of) an SMT-LIB compatible interface and can either be used by a generic SMT-LIB parser, or an external tool. This structure is shown in the following picture.

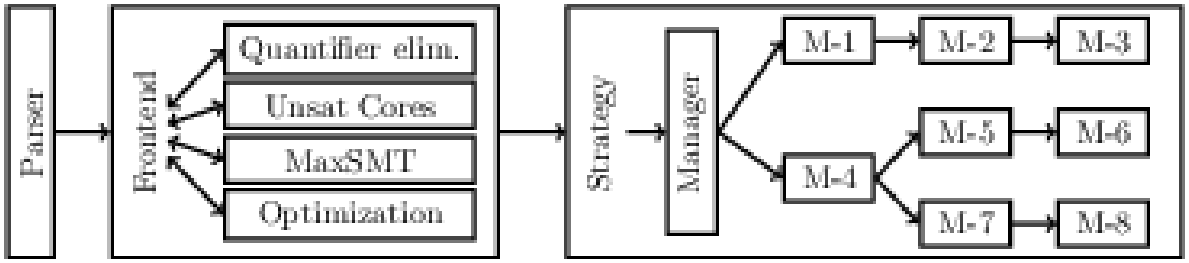

Figure 1 System architecture

The parser itself is implemented in `cli/parser/` and run from `cli/tools/execute_smtlib.h`. The template argument `Executor` is usually instantiated with the executor from `cli/tools/Executor.h` which corresponds to the frontend. The components in the frontend are taken from the respective SMT-RAT libraries.

The manager is a generic class from `smtrat-solver/Manager.h` that every strategy (from `smtrat-strategies/`) inherits from and only constructs the strategy graph in its constructor. The strategy graph is at the core of the composition of SMT-RAT modules and the following picture shows how a single module is embedded in a strategy.

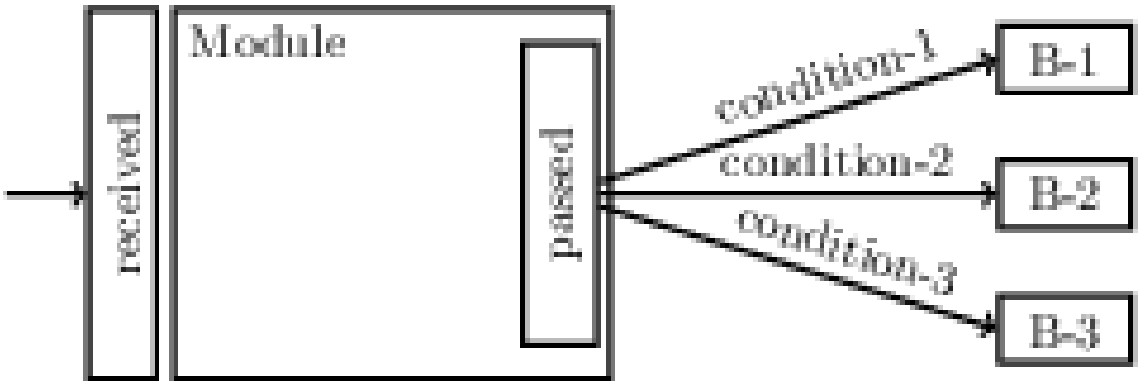

Figure 2 Module in a strategy

Every module has (a pointer to) a set of received formulae that represent its input and a set of passed formulae that represent the formula that is passed on to some backend. The module may ”solve” the query from its input on its own, or it may pass (one or more) queries to its backends (in this case B-1, B-2 and B-3). Arrows to backends may be labeled with conditions that restrict whether this particular backend can be used, for example checking whether the passed formulae are linear, contain integer variables or bit-vector formulae.

When a module issues a backend call the manager identifies all suitable backend modules (where the condition evaluates to `true`) and calls all backend modules on the passed formulae. This happens either sequentially (until one backend module solves the query) or in parallel.

3.3 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(\f$\varphi\f$)` (`remove(\f$\varphi\f$)`). 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.

3.4 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, \eg whether it contains equations, or about the polynomials, e.g., whether they are linear. Furthermore, each edge carries a unique priority value from $\Sigma = \{1, \ldots, |E|\}$.

3.5 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 (σ, 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.

3.6 Procedures implemented as modules

The heart of an SMT solver usually forms a SAT solver. In SMT-RAT, the module SATModule abstracts the received formulae to propositional logic and uses the efficient SAT solver MiniSat **[minisat]** to find a Boolean assignment of the abstraction. It invokes `runBackends` where the passed formulae of the SATModule contain the constraints abstracted by the assigned Boolean variables in a less-lazy fashion **[sebastiani2007lazy]**. [Todo: Make a concise description here, refer to extensive discussion of modules]

3.7 Infeasible subsets and lemmas

Infeasible subsets and lemmas, which contain only formulas from $C_{pas}(SATModule)$ of a preceding SATModule, 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 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 sec↩ :auxfunctions. 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.

4.1 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.
- `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 FormulaT 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 sec:runbackend.

4.2 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.

4.2.1 Informing about a constraint

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

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`.

4.2.2 Adding a received formula

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

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.

4.2.3 Removing a received formula

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

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.

4.2.4 Checking for satisfiability

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

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*.

4.2.5 Updating the model/satisfying assignment

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

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.

4.3 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 chapter:composingats. 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:

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

Adds the formula at the given posititon 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.

```
pair<ModuleInput::iterator,bool> addSubformulaToPassedFormula(const Formula&)
pair<ModuleInput::iterator,bool> addSubformulaToPassedFormula(const Formula&, const Formula&)
pair<ModuleInput::iterator,bool> addSubformulaToPassedFormula(const Formula&, shared_ptr<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. \end{itemize} 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`.

4.4 Auxiliary 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`.

- `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.

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, \eg \supportedLogics, 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.

In Section sec::strategy we have already introduced a strategy and in the following of this chapter we give a brief introduction to the existing modules equipped with an estimation of their input-based performances.

4.5 Existing module implementation

Available modules

4.6 Available modules

SMT-RAT comes with the following modules that are documented in some more detail below:

BEModule BVModule CNFerModule CSplitModule CoCoAGBModule CubeLIAModule CurryModule EMModule ESModule FPPModule FouMoModule GBModule GBPPModule ICEModule ICPModule IncWidthModule IntBlastModule IntEqModule LRAModule LVEModule MCBModule NRAILModule NewCADModule NewCovering↩ Module PBGaussModule PBPPModule PFEModule SATModule STropModule SplitSOSModule SymmetryModule UFCegarModule UnionFindModule VSModule

4.6.1 BEModule

4.6.2 BVModule This module implements an encoder for bitvector formulae to propositional logic. It is described in more detail in [**?**].

4.6.3 CNFerModule Transforms its received formula into conjunctive normal form CNF.

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

4.6.4 CSplitModule Implements solving for nonlinear integer arithmetic using incremental linearization. This module was implemented in [**?**] based on [**?**].

4.6.5 CoCoAGBModule Uses Gröbner bases for theory solving, very much like smtrat::GBModule. However, for the underlying implementation of Gröbner bases, this module does not use carl::groebner but CoCoALib.

4.6.6 CubeLIAModule Implements cube-based tests for linear integer arithmetic based on [**?**].

4.6.7 CurryModule Implements curryfiyng preprcessing as described in [**?**].

4.6.8 EMModule This module addempts to eliminate multiple factors from equations and inequations.

4.6.9 ESModule Uses equations (or Boolean assertions) to eliminate variables from the remaining formula. Let the formula have the form $e \wedge \varphi$', then we use knowledge gained from e to simplify φ'. If e is an arithmetic equation such that we can rewrite it to the form $x = t$ (with x a variable) then we substitute t for x into φ'. Otherwise we simply replace e with $true$ in φ'. This is done recursively in the formula.

4.6.10 FPPModule This module implements a generic preprocessing facility. It runs a given strategy on the input and retrieves the simplified formula using the facilities of the Manager class. This process is iterated until a fix point is reached – no further simplification was done by the strategy – or a predefined number of iterations was performed. The resulting simplified formula is then forwarded to the backend.

The strategy should be a linear sequence of preprocessing modules, as passing simplified formulas back to the caller is not possible in a meaningful way for general Module classes.

4.6.11 FouMoModule Implements the SMT compliant Fourier-Motzkin algorithm. 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 e.g. in the case of a monomial x^i (where i is a positive integer) only occuring in the shape of this monomial. Such a monomial is subsequently eliminated as common linear variables are eliminated. One can tune a threshold parameter in order to determine when, regarding the size of the considered constraints, this module shall call the backends. In the latter case, the backends are called with the constraint set that is obtained after eliminating a certain number of variables.

4.6.11.1 Integer arithmetic One can also use this approach for (linear) integer arithmetic as unsatisfiability over the real domain implies unsatisfiability over the integer domain. In addition to that, one can heuristically try to construct integer solutions by considering the lowest upper and the highest lower bound of a variable that can be derived from the respective elimination step. Note that this approach is incomplete.

4.6.12 GBModule Implements the Gröbner bases based procedure as presented in [**?**]. 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.

4.6.12.0.1 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.

4.6.13 GBPPModule Uses Gröbner basis computations to simplify the input formula. The underlying implementation of Gröbner bases is used from carl::groebner.

The fundamental idea is as follows: Separate the input formula into equalities and inequalities and compute the Gröbner basis of the equalities. Now we can replace the equalities with the Gröbner basis (if this "seems easier") and also simplify the inequalities using the Gröbner basis.

4.6.14 ICEModule This module tries to find simple chains of inequalities that can be combined to form a cycle. It converts constraints of the form $x \geq y + z$ to a hypergraph (with an edge going from x to y, z) and uses the coefficients as edge weights. If this hypergraph contains cycles, these can be used to infer additional constraints on the individual variables. In particular, zero cycles induce equality of variables while negative cycles reveal conflicts.

4.6.15 ICPModule Implements a combination of interval constraint propagation equipped with a Newton-based contraction [**?**] and LRA solving, for which we use our LRAModule. The implementation is inspired by [**?**], 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 [**?**]. 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.

4.6.15.0.1 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.

4.6.16 IncWidthModule This module is meant to be used for solving nonlinear integer arithmetic problems by encoding them into bitvector arithmetic formulas (as done in smtrat::IntBlastModule), as described in [**?**] and [**?**], both (heavily) inspired by [**?**]. This approach heavily benefits from knowing bounds on individual variables, both in terms of its ability to find infeasibility and its performance in general.

In this module, we use smtrat::ICPModule to infer new bounds and call the backend incrementally with growing bounds until the computed upper bounds are met.

4.6.17 IntBlastModule This module implements encoding nonlinear integer arithmetic problems into bitvector arithmetic formulas as described in [**?**] and [**?**], both (heavily) inspired by [**?**].

4.6.18 IntEqModule Implements the SMT compliant equation elimination method presented in [**?**]. Hence, this module either determines that the equations of an instance are unsatisfiable or calculates a certain number of substitutions for variables. The latter can be used to eliminate the equations by substituting every variable for which a substitution exists by its substitution. The set of constraints that we obtain with this procedure, not containing equations anymore, is passed to the backends.

4.6.19 LRAModule Implements the SMT compliant simplex method presented in [**?**]. 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 smtrat::LRAModule might need to communicate a lemma/tautology to a preceding smtrat::SATModule, if it receives a constraint with the relation symbol $\neq$ and the strategy needs for this reason to define a smtrat::SATModule at any position before an smtrat::LRAModule.

4.6.19.0.1 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 [**?**]. It is also supported to combine these approaches. Note that for all of them the smtrat::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 smtrat::LRAModule.

4.6.19.0.2 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.

4.6.20 LVEModule This module aims to eliminate lone variables that only occur once. We call a variable a lone variable if it only occurs in a single constraint. Furthermore we only try to eliminate the variable if this constraint is a top-level constraint.

4.6.21 MCBModule This module attempts to detect logic structures that essentially encode a multiple-choice on an arithmetic variable. It extracts this information and translates it to a Boolean structure, thereby eliminating the arithmetic variable.

4.6.22 NRAILModule This module implements incremental linearization as described in [**?**], roughly following [**?**] and [**?**]. In addition, it implements an ICP-based axiom instantiation.

4.6.23 NewCADModule This module is based on the implementation in smtrat::cad. It is described in [**?**] in some detail.

4.6.24 PBGaussModule This module tries to simplify pseudo-Boolean problems using Gauss as described in [**?**].

4.6.25 PBPPModule This module implements a variety of preprocessing techniques for pseudo-Boolean problems. Many of them are based on [**?**].

4.6.26 PFEModule Implements factor elimination in polynomials based on factorization and variable bounds.

Given a constraint $p \sim 0$ with $\sim \in \{=, \neq, \geq, >, \leq, <\}$ and bounds b on the variables in p, the module does the following: It computes a factor q such that $p = q \cdot r$ and evaluates q on the intervals represented by the bounds. If the resulting interval $q(b)$ if sign-invariant, the constraint can be simplified or additional constraints can be added. We consider an interval to be sign-invariant, if it is positive, semi-positive, zero, semi-negative or negative.

4.6.26.0.1 Simplifications The following simplifications are done for for $q \cdot r \sim 0$:

$\sim$	$q(b) > 0$	$q(b) \geq 0$	$q(b) = 0$	$q(b) < 0$	$q(b) \leq 0$
$=$	$p := r$	$f := q = 0 \vee r = 0$	$p := 0$	$p := r$	$f := q = 0 \vee r = 0$
$\neq$	$p := r$	$f := q > 0 \wedge r \neq 0$	$p := 0$	$p := r$	$f := q < 0 \wedge r \neq 0$
$\geq$	$p := r$	$f := q = 0 \vee r \geq 0$	$p := 0$	$c := r \leq 0$	$f := q = 0 \vee r \leq 0$
$>$	$p := r$	$f := q > 0 \wedge r > 0$	$p := 0$	$c := r < 0$	$f := q < 0 \wedge r < 0$
$\leq$	$p := r$	$f := q = 0 \vee r \leq 0$	$p := 0$	$c := r \geq 0$	$f := q = 0 \vee r \geq 0$
$<$	$p := r$	$f := q > 0 \wedge r < 0$	$p := 0$	$c := r > 0$	$f := q < 0 \wedge r > 0$

Notation: $p := p$' replaces the polynomial, $c := p' \sim 0$ replaces the whole constraint by a new one and $f := f$' replaces the constraint by a new formula.

To maximize the cases where $q(b)$ actually is sign-invariant, we proceed as follows. We compute a factorization of p, that is a number of polynomials p_i such that $p = \prod_{i=1}^{k} p_i$. We now separate all p_i into two sets: P_q for all sign-invariant p_i and P_r for all other p_i. Thereby, we set $q = \prod_{t \in P_q} t$ and $r = \prod_{t \in P_r}$ and know that $q(b)$ is sign-invariant.

Instead of computing $q(b)$ once again afterwards, we can determine the type of sign-invariance of $q(b)$ from the types of sign-invariances of the factors from P_q. Assume that we start with a canonical factor 1 and a sign-invariance of $>$, we can iteratively combine them like this:

4.6.26.0.2 Combining types of sign-invariance Combining two types of sign-invariance is done as follows:

$\cdot$	$>$	$\geq$	$=$	$\leq$	$<$
$>$	$>$	$\geq$	$=$	$\leq$	$<$
$\geq$	$\geq$	$\geq$	$=$	$\leq$	$\leq$
$=$	$=$	$=$	$=$	$=$	$=$
$\leq$	$\leq$	$\leq$	$=$	$\geq$	$\geq$
$<$	$<$	$\leq$	$=$	$\geq$	$>$

4.6.27 SATModule This module abstracts it's received formula, being any SMT formula of the supported logics of \smtrat, 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∼**[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.

4.6.27.0.1 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.

4.6.28 STropModule Implements the subtropical satisfiability method as described in [**?**] based on [**?**].

4.6.29 SplitSOSModule Splits the left hand side of constraints according to their sum-of-squares decomposition.

4.6.30 SymmetryModule This module tries to recognize syntactic symmetries in the input formula and adds symmetry breaking constraints. The core functionality is provided by CArL through carl::formula::breakSymmetries() which internally encodes the formula as a graph and uses bliss to find automorphisms on this graph.

4.6.30.0.1 Efficiency Finding automorphisms is as difficult as determining whether two graphs are isomorphic, and it is not known whether this problem can be solved in polynomial or exponential time. In practice, current solvers like bliss perform very good on large graphs and we therefore assume this module to be sufficiently fast.

4.6.31 UFCegarModule This module implements a CEGAR-approach for solving uninterpreted functions (or rather reducing uninterpreted functions to equality logic).

4.6.32 UnionFindModule This module implements a theory solver for equality logic based on a persistent union find structure (based on the immer library).

4.6.33 VSModule Implements the virtual substitution method for a conjunction of constraints as described in [**?**]. 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.

4.6.33.0.1 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.

5 Strategies

To compose modules to a solver, strategies are used. Strategies are represented as trees of modules where the input file is passed to the root module. Every module can issue calls to its **backend modules** (via `runBackends`) which then calls its child modules in the strategy on the formula this module has put in the passed formulas. If no backend module exists, the call returns `unknown`.

A Strategy is specified by deriving from the Manager class and settings its strategy in the constructor using the `setStrategy` member function. A simple example, supporting only propositional logic, would look like this:

```
class PureSAT: public Manager {
    public:
    PureSAT(): Manager() {
        setStrategy(
            addBackend<SATModule<SATSettings1>>()
        );
    }
};
```

Each module is added to the strategy tree using `addBackend<Module class>` where the module class is usually a template that is instantiated with some settings. A slightly more complex, where the SATModule has the NewCADModule available for theory calls, might look as follows:

```
class CADOnly: public Manager {
    public:
    CADOnly(): Manager() {
        setStrategy(
            addBackend<SATModule<SATSettings1>>(
                addBackend<NewCADModule<NewCADSettingsFOS>>()
            )
        );
    }
};
```

Note that the arguments to `setStrategy` and `addBackend` can not only be a single element, but an `initializer_list` of multiple backends as well. In this case the backends are called in order until one returns a conclusive result (that is: not `unknown`). If SMT-RAT was configured to allow for parallel execution, the backends are executed in parallel and the result of the first backend to terminate is used.

```
class CADVSICP: public Manager {
    public:
    CADVSICP(): Manager() {
        setStrategy(
            addBackend<SATModule<SATSettings1>>({
                addBackend<NewCADModule<NewCADSettingsFOS>>(
                    addBackend<VSModule<VSSettings234>>()
                ),
                addBackend<ICPModule<ICPSettings1>>()
            })
        );
    }
};
```

Finally, we can also attach conditions to individual backends to restrict their applicability. For example, we can construct a strategy that uses separate sub-strategies for linear and nonlinear problems:

```
class RealSolver: public Manager {
    static bool is_nonlinear(carl::Condition condition) {
        return (carl::PROP_CONTAINS_NONLINEAR_POLYNOMIAL <= condition);
    }
    static bool is_linear(carl::Condition condition) {
        return !(carl::PROP_CONTAINS_NONLINEAR_POLYNOMIAL <= condition);
    }
    public:
    RealSolver(): Manager() {
        setStrategy({
            addBackend<FPPModule<FPPSettings1>>(
                addBackend<SATModule<SATSettings1>>(
                    addBackend<ICPModule<ICPSettings1>>(
                        addBackend<VSModule<VSSettings234>>(
                            addBackend<NewCADModule<NewCADSettingsFOS>>()
                        )
                    )
                )
            ).condition( &is_nonlinear ),
            addBackend<FPPModule<FPPSettings1>>(
                addBackend<SATModule<SATSettings1>>(
                    addBackend<LRAModule<LRASettings1>>()
                )
            ).condition( &is_linear )
        });
    }
};
```

6 Using SMT-RAT

SMT-RAT can be used in two ways, either as a standalone solver or as a C++ library within some other software.

6.1 Standalone solver

Before actually compiling SMT-RAT into a binary, an appropriate strategy should be selected. While a number of strategies are available, it sometimes makes sense to craft a strategy for the specific problem at hand. Please refer to Strategies on how to create strategies. To select a strategy use `ccmake` to set the `SMTRAT_Strategy` variable accordingly and then build the solver binary using `make smtrat-shared`. Use `./smtrat-shared --strategy` to check whether the desired strategy is used.

The solver binary can now be used to solve input in the `smt2` format, either given as a filename or on standard input:

```
./smtrat-shared <input file>
cat <input file> | ./smtrat-shared -
```

Note that the solver binary can perform many other tasks as well that we discuss below. Some of these are enabled (or disabled) by a set of `cmake` options of the form `CLI_ENABLE_*` and the currently available ones can be obtained as follows:

```
$ ./smtrat-shared --help
Usage: ./smtrat-shared [options] input-file

Core settings:
  --help                                show help
  --info                                show some basic information about this
                                        binary
  --version                             show the version of this binary
  --settings                            show the settings that are used
  --cmake-options                       show the cmake options during
                                        compilation
  --strategy                            show the configured strategy
  --license                             show the license
  -c [ --config ] arg                   load config from the given config file
```

```
Solver settings:
  --preprocess                          only preprocess the input
  --pp-output-file arg                  store the preprocessed input to this
                                        file
  --to-cnf-dimacs                       transform formula to cnf as dimacs
  --to-cnf-smtlib                       transform formula to cnf as smtlib
  --print-model                         print a model if the input is
                                        satisfiable
  --print-all-models                    print all models of the input

Validation settings:
  --validation.export-smtlib            store validation formulas to smtlib
                                        file
  --validation.smtlib-filename arg (=validation.smt2)
                                        filename of smtlib output
  --validation.channel arg              add a channel to be considered


Module settings:
  --module.parameter arg                add a parameter for modules (key=value)

Parser settings:
  --dimacs                              parse input file as dimacs file
  --opb                                 parse input file as OPB file
  --input-file arg                      path of the input file
  --disable-uf-flattening               disable flattening of nested
                                        uninterpreted functions
  --disable-theory                      disable theory construction

Analysis settings:
  --analyze.enabled                     enable formula analyzer
  --analyze.projections arg (=none)     which CAD projections to analyze (all,
                                        collins, hong, mccallum,
                                        mccallum_partial, lazard, brown, none)

CAD Preprocessor settings:
  --cad.pp.no-elimination               disable variable elimination
  --cad.pp.no-resultants                disable resultant rule
```

6.1.1 Formula analysis One sometimes wants to only obtain certain information about the given formula, usually for statistical purposes. SMT-RAT exposes a formula analyzer that gives a number of properties of the formula.

```
$ ./smtrat-shared --analyze.enabled <input file>
```

6.1.2 Preprocessing While many SMT-RAT strategies employ certain preprocessing techniques, it is sometimes convenient to apply this preprocessing ahead of time, for example to normalize the inputs. The result is either printed or written to an output file if `--pp-output-file` is given.

```
$ ./smtrat-shared --preprocess --pp-output-file <output file> <input file>
```

6.1.3 Quantifier elimination Instead of regular SMT solving, SMT-RAT can also perform quantifier elimination tasks as described in **[Neuhaeuser2018]**. This technique is used when the SMTLIB file contains a `eliminate-quantifiers` command like `(eliminate-quantifiers (exists x y) (forall z))`.

6.1.4 DIMACS solving For purely propositional formulae (i.e. SAT solving) one usually uses the (much more compact) DIMACS format instead of SMT-LIB. Note that SMT-RAT still uses the configured strategy to solve the given input instead of only a SAT solver, allowing to use custom preprocessing techniques.

```
$ ./smtrat-shared --dimacs <input file>
```

6.1.5 Pseudo-Boolean solving Another interesting solving task is concerned with pseudo-Boolean formulae where Boolean variables are used in arithmetic constraints (and implicitly considered as being integers over just zero and one). A special input format called OPB exists (rather similar to DIMACS) that can be read and solved with a strategy based on [**?**] as follows:

```
$ ./smtrat-shared --opb <input file>
```

6.1.6 Optimization For many applications, one wants not only some feasible solution but rather an optimal solution with respect to some objective function. This is used when the SMTLIB file contains one or more `(minimize)` or `(maximize)` commands with semantics similar to what is described for `z3`.

6.2 Embedding in other software

Instead of using SMT-RAT as a standalone solver, it can also be embedded in other software.

6.2.1 Interface The easiest way is to embed a certain strategy.

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();
```

As all strategies are derived from the Manager class, we can thus use the Manager interface. The most important methods are the following:

- `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 `SAT`, if satisfiability has been detected, or `UNSAT`, if the formulas are not satisfiable, and `UNKNOWN`, if the composition cannot give a conclusive answer. If the answer has been `SAT`, we get the model, satisfying the conjunction of the given formulas, using `model()` and, if it has been `UNSAT`, 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 QF_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 `SATModule` before any theory module is used (It is possible to define a strategy using conditions in a way, that we achieve an SMT solver, even if for some cases no `SATModule` is involved before a theory module is applied.) and, therefore, the SMT solver never communicates these lemmas as they are already processed by the `SATModule`. A better explanation on the modules and the strategy can be found in `system architecture`. 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.

- `std::vector<FormulaT> lemmas() const` Returns valid formulas, which we call lemmas. For instance the `ICPModule` 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 and ... and c_n) -> (p <= r or p > r)` where c_1, .., 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 being a rational number. Hence, splitting decisions always form a tautology. We recommend to use the `ICPModule` only in strategies with a preceding `SATModule`. The same holds for the `LRAModule`, `VSModule`, and `CADModule` if used on QF_NIA formulas. Here, again, splitting decisions might be communicated.

Of course, the Manager interface contains more methods that can be found at Manager.

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

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

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 multivariate polynomial with 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.

6.2.3 Boolean combinations of constraints and Boolean variables For more information, check out the docs of `CArL`.

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 FormulaT class, 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 `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 $\wedge$ and $\vee$ are n-ary constructors, $\neg$ is a unary constructor and all the other Boolean operators are binary.

6.2.4 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 \polynomialOrder 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 simplifications, 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.

6.2.5 Linking [Todo: example how linking works]

7 Other tools

7.1 Benchmarking

Alongside SMT-RAT, `benchmax` allows to easily run solvers on benchmarks and export the results. See Benchmax for more information.

7.2 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. Usually, as the ultimate goal is a minimal example that triggers some bug by the application of a set of transformation rules.

7.2.1 SMT-RAT's own delta tool SMT-RAT has its own tool for delta debugging, see Delta for more information.

7.2.2 Using ddSMT Currently, we recommend using `ddSMT` as the state-of-the-art tool for delta debugging.

For a complete guide, we refer to the documentation of ddSMT.

Here, we give instructions for the most common use cases when developing with SMT-RAT. There are mainly two common scenarios where we use delta debugging: If there is a failing assertion or if SMT-RAT returns a wrong result. For these scenarios, we provide wrapper scripts preparing the input to ddsmt properly.

First, install `ddSMT` and `z3`:
```
pip3 install ddsmt
sudo apt install z3
```

In case of a *failing assertion* on `in_file.smt2`, run
```
/src/smtrat/utilities/ddsmt/ddsmt-assertion in_file.smt2 out_file.smt2 path_to_solver
```

In case SMT-RAT returns a *wrong result* on `in_file.smt2`, run
```
/src/smtrat/utilities/ddsmt/ddsmt-wrong in_file.smt2 out_file.smt2 path_to_solver
```

7.3 Analyzer

The `smtrat-analyzer` library can analyze static properties input formulas (such as number of variables, degrees, CAD projections, ...).

To use it from the CLI, build smtrat using `CLI_ENABLE_ANALYZER=ON` and `SMTRAT_DEVOPTION_Statistics=ON`. A single input file can be analyzed by running `./smtrat --analyze.enabled --stats.print input.smt2`; properties will be printed as statistics object, note that the solver will not be called. For further options, see `./smtrat --help`.

To collect properties of all formulas of a benchmark sets, the analyzer can be used with `benchmax`. To do so, specify add the binary as an `smtrat-analyzer` solver; for more information, see the `benchmax` subpage.

7.4 Preprocessing

WIP

7.5 Benchmax

Benchmax is a tool for automated benchmarking. Though it was developed and is primary used for testing SMT solvers, it aims to be agnostic of the tools and formats as good as possible.

Its fundamental model is to load a list of tools and a list of benchmarks, run every tool with every benchmark, obtain the results and output all the results. While the benchmarks are fixed to a single file, we allow to choose between different tool interfaces, execution backends and output formats.

7.5.1 General usage Benchmax could be called as follows:
```
./benchmax -T 1m -M 4Gi -S /path/to/smtrat-static -X out.xml -b ssh --ssh.node user@127.0.0.1@5\#9 -D
 /path/to/benchmark/set
```

This will run benchmax with the SMT-RAT tool at `/path/to/smtrat-static` on the `/path/to/benchmark/set` with a time limit of 1 minute and a memory limit of 4 gigabyte. The resulst will be written to `out.xml`. The execution is done on an SSH backend on localhost with 9 connections and executing 5 parallel jobs per connection.

It is recommended to use static builds to avoid issues with missing libraries.

For more information, run `./benchmax --help`.

7.5.1.1 Collecting statistics Some tools like SMT-RAT or Z3 can provide statistics about the solving process for each individual benchmark. By using the `-s` respectively `--statistics`, statistics are collected and stored in the output file.

7.5.1.1.1 Large output It is recommended to aggregate statistics as much as possible inside SMT-RAT. However, if the output might get large, you might want to use `--use-tmp` to prevent benchmax running out of memory.

7.5.1.2 Working with the results In the SMT-RAT repository, two utilites for converting the result XML file are included:

- `utilities/benchmax/xml2ods.py` for converting it to a *Flat XML LibreOffice Calc Sheet*.
- An XML filter `utilities/benchmax/OOCImporter.xsl` for converting it to a *Flat XML LibreOffice Calc Sheet*.
- `utilities/benchmax/evaluation` is a small python library for importing the results into Python (or a Jupyter notebook), inspecting the results and preparing plots. For more information, see Benchmax python utility.

7.5.2 Tools A tool represents a binary that can be executed on some input files. It is responsible for deciding whether it can be applied to some given file, building a command line to execute it and retrieve additional results from `stdout` or `stderr`.

The generic tool class `benchmax::Tool` can be used as a default. It will run the given binary on any input file and benchmax records the exit code as well as the runtime.

If more information is needed a new tool class can be derived that overrides or extends the default behaviour. Some premade tools are available (and new ones should be easy to create):

- benchmax::MathSAT
- benchmax::Minisatp
- benchmax::SMTRAT
- benchmax::SMTRAT_OPB
- benchmax::Z3

7.5.3 Backends A backend offers the means to run the tasks in a specific manner. A number of different backends are implemented that allow for using benchmax in various scenarios.

7.5.3.1 LocalBackend The LocalBackend can be used to execute benchmarks on a local machine easily. It does not allow for any parallelization but simply executes all tasks sequentially.

7.5.3.2 SlurmBackend The SlurmBackend employs the Slurm workload manager to run benchmarks on a cluster. It essentially collects all tasks that shall be run in a file and creates an appropriate slurm submit file. It then submits this job, waits for it to finish and collects the results from the output files.

The Slurm backend supports starting (`--mode execute`) and collecting results (`--mode collect`) separately, thus avoiding the need for running benchmax in a screen.

7.5.3.3 SSHBackend The SSHBackend can be used if you have multiple workers that can be reached via ssh (essentially a cluster without a batch job system). It allows to configure one or more computing nodes and manually schedules all the jobs on the different computing nodes.

Using `--ssh.node`, a node is specified in the format `<user>[:<password>]@<hostname>[:<port = 22>][@<cores = 1>][#<connections = 1>]`; benchmax will then connect to the specified server with given credentials `connections` times and will run `cores` threads per connection. Thus, the number of overall parallel threads equals `connections * cores` (note that enough memory should be available on the node).

By setting the flag `--ssh.resolvedeps`, dependencies (i.e. dynamic libraries) are resolved and uploaded with the solver.

7.5.3.4 CondorBackend (unstable) The CondorBackend is aimed at the HTCondor batch system and works similar to the SlurmBackend. Note however that this backend is not well tested and we therefore consider it unstable.

7.5.4 Troubleshooting

- If benchmax terminates due to a segmentation fault without any message, then it probably exceeded available memory. A common error is that SMT-RAT has produced too much logging output. To fix this, simply build SMT-RAT with `LOGGING=OFF`.

7.5.5 Benchmax python utility This tool allows loading XML files from Benchmax into a pandas dataframe, inspecting the results and visualizing them.

This is useful for working with the results in a Jupyter notebook.

Dependencies:

- pandas
- matplotlib
- pillow
- numpy
- tikzplotlib

```
pip3 install pandas matplotlib tikzplotlib numpy pillow
```

7.5.5.1 Loading XMLs into a pandas dataframe First, install this directory as python library; e.g. on Ubuntu

```
cd  /.local/lib/python3.8/site-packages/ # path to your python site-packages directory
ln -s  /src/smtrat/utilities/benchmax/ # path to the benchmax utility
```

In your Jupyter notebook:

```
import benchmax.evaluation as ev
df = ev.xml_to_pandas("path_to/stats_file.xml", {"smtrat-static": "solver_name"},
 ["statistics_name_1","statistics_name_2"]) # second and third parameter is optional
```

or, to load multiplle XMLs into one:

```
import benchmax.evaluation as ev
df = ev.xmls_to_pandas({"path_to/stats_file_1.xml": {"smtrat-static": "solver_name_1"},
 "path_to/stats_file_2.xml": {"smtrat-static": "solver_name_2"}}, ["statistics_name_1","statistics_name_2"]) #
 second parameter is optional
```

This will create a dataframe with columns `(solver_name_1, "answer")`, `(solver_name_1, "runtime")`, `(solver_name_1, "statistics_name_1")` etc (as multi-index).

7.5.5.2 Computing a virtual best solver A virtual best is a solver that behaves optimal on each input w.r.t. a set of solvers. It is computed by selecting for each benchmark instance the solver with shortest running time.

To compute the virtual best solver named `VB` w.r.t. `solver1`,`solver2` and `solver3` and considering statistics `statistics_name_1`.

```
df = df.join(ev.virtual_best(df, ["solver1","solver2","solver3"], "VB", ['statistics_name_1']))
```

7.5.5.3 Plotting We provide the `performance_profile` and `scatter_plot` functions to easily generate a performance profiles and scatter plots.

7.5.5.4 Show a summary of an XML file There are several methods for quickly inspecting the results of a run provided. For instance, the following script can be used to view a summary of a singel XML file and to show instances with wrong results or segmentation faults:

```
#!/usr/bin/env python3
import benchmax.evaluation as ev
import sys
df = ev.xml_to_pandas(sys.argv[1])
ev.inspect(df)
```

SMT-RAT provides a small utility script for showing a summary of one or more XML files:

```
 /Code/smtrat/utilities/benchmax/view.py result_1.xml result_2.xml
```

7.5.5.5 Exporting data to Latex

7.5.5.5.1 Pandas dataframe to latex table `df.to_latex(buf='file.tex')`

7.5.5.5.2 matplotlib to tikz plot see `tikzplotlib`

7.6 Delta

7.6.1 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 [**?**] 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 SMT-LIB files. It has some knowledge of the semantics of the corresponding logics, but only operates on nodes. Any SMT-LIB 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.

7.6.1.1 Using delta `delta` is currently contained in the SMT-RAT reopsitory and can be built using `make delta`.

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`.

7.6.1.1.1 Exit code As delta relies on the exit code of the program, make sure that this event results in a unique exit code:

- **Specific assertions**: Return a unique exit code by replacing the assertion with `if (!assertion_condition) { std:exit(70); }`.
- **Faulty output**: Remove the `(set-info :status unsat)/(set-info :status sat)` line from the benchmark and use the script `utilities/result-incorrect.py` as solver. Note that you need to install z3 first.

8 Developers information

- Code style
- Documentation
- Settings
- Logging
- Statistics and timing
- Testing
- Validation
- Checkpoints
- Finding and Reporting Bugs
- Tools

8.1 Code style

8.1.0.1 Code formatting ClangFormat allows to define code style rules and format source files automatically. A .clang-format file is provided with the repository. Please use this file to format all sources.

8.1.0.2 Naming conventions For all new code, the following rules apply.

- type names and template parameter: CamelCase
- variable and function names: snake_case
- compiler macros and defines: ALL_UPPERCASE
- enum values: UPPERCASE
- (private) class members: start with m_ respectively mp_ for pointers and mr_ for references

8.1.0.3 Headers and namespaces The following conventions apply for new modules added to SMT-RAT and CArL. When adding code to existing structures, be consistent with existing code.

- Every file should represent a *module*.
- Every (sub-)folder has its own namespace.
- Namespaces and modules have names in camel_case.

8.1.0.4 C++ features

- As of now, please stick to C++17 features.
- Use enum class instead of enum.

8.2 Documentation

For SMT-RAT, there are two sources of documentation: the in-source API documentation and the more manual-like documentation (you are reading right now). While the in-source API documentation is generated based on the doxygen comments from the actual source files, the manual is generated from the markdown files in doc/markdown/.

8.2.0.1 Writing documentation Note that some of the documentation may be incomplete or rendered incorrectly, especially if you use an old version of doxygen. Here is a list of known problems:

- Comments in code blocks (see below) may not work correctly (e.g. with doxygen 1.8.1.2). See here for a workaround. This will however look ugly for newer doxygen versions, hence we do not use it.
- Files with static_assert statements will be incomplete. A patch is pending and will hopefully make it into doxygen 1.8.9.
- Member groups (usually used to group operators) may or may not work. There still seem to be a few cases where doxygen messes up.
- Documenting unnamed parameters is not possible. A corresponding ticket exists for several years.

8.2.0.2 Literature references Literature references should be provided when appropriate by citing references from the bibtex database located at doc/literature.bib using the @cite command. The labels are the last name of the first author and the four-digit year. In case of duplicates, we append lowercase letters.

These references can be used with @cite label, for example like this:

```
/**
 * Checks whether the polynomial is unit normal
 * @see @cite GCL92, page 39
 * @return If polynomial is normal.
 */
bool isNormal() const;
```

8.2.1 Code comments

8.2.1.1 File headers

```
/**
 * @file <filename>
 * @ingroup <groupid1>
 * @ingroup <groupid2>
 * @author <author1>
 * @author <author2>
 *
 * [ Short description ]
 */
```

Descriptions may be omitted when the file contains a single class, either implementation or declaration.

8.2.1.2 Namespaces

Namespaces are documented in a separate file, found at '/doc/markdown/codedocs/namespaces.dox'

8.2.1.3 Class headers

```
/**
 * @ingroup <groupid>
 * [ Description ]
 * @see <reference>
 * @see <OtherClass>
 */
```

8.2.1.4 Method headers

```
/**
 * [ Usage Description ]
 * @param <p1> [ Short description for first parameter ]
 * @param <p2> [ Short description for second parameter ]
 * @return [ Short description of return value ]
 * @see <reference>
 * @see <otherMethod>
 */
```

These method headers are written directly above the method declaration. Comments about the implementation are written above the or inside the implementation.

The `see` command is used likewise as for classes.

8.2.1.5 Method groups

There are some cases when documenting each method is tedious and meaningless, for example operators. In this case, we use doxygen method groups.

For member operators (for example `operator+=`), this works as follows:

```
/** @name In-place addition operators
 *  @{ */
/**
 * Add something to this polynomial and return the changed polynomial.
 * @param rhs Right hand side.
 * @return Changed polynomial.
 */
MultivariatePolynomial& operator+=(const MultivariatePolynomial& rhs);
MultivariatePolynomial& operator+=(const Term<Coeff>& rhs);
MultivariatePolynomial& operator+=(const Monomial& rhs);
MultivariatePolynomial& operator+=(Variable::Arg rhs);
MultivariatePolynomial& operator+=(const Coeff& rhs);
/// @}
```

8.2.2 Writing out-of-source documentation

Documentation not directly related to the source code is written in Markdown format, and is located in `/doc/markdown/`.

8.3 Settings

As described in sec::modules, each module has a settings template parameter.

8.3.0.1 Dynamic settings

These settings are considered at compile time. For certain purposes, it is unhandy to recompile SMT-RAT for every parameter. For this, module parameters can be set

- via the command line using `--module.parameter key1=value1 --module.parameter key2=value2 ...` or
- via the `set-option` command of SMT-LIB: `(set-option :key "value")` (note that you need to pass every value as a string, even if it represents a number!).

These parameters can be accessed (most appropriately in a settings object of a module) via `int my_setting = settings_module().get("my_module.my_setting", 2)` or `std::string my_setting = settings_module().get("my_module.my_setting", "default_value")`. Note that the second parameter is the default value also specifying the type of the setting to which it is parsed.

8.4 Logging

8.4.0.1 Logging frontend The frontend for logging is defined in logging.h.

It provides the following macros for logging:

- SMTRAT_LOG_TRACE(channel, msg)
- SMTRAT_LOG_DEBUG(channel, msg)
- SMTRAT_LOG_INFO(channel, msg)
- SMTRAT_LOG_WARN(channel, msg)
- SMTRAT_LOG_ERROR(channel, msg)
- SMTRAT_LOG_FATAL(channel, msg)
- SMTRAT_LOG_FUNC(channel, args)
- SMTRAT_LOG_FUNC(channel, args, msg)
- SMTRAT_LOG_ASSERT(channel, condition, msg)
- SMTRAT_LOG_NOTIMPLEMENTED()
- SMTRAT_LOG_INEFFICIENT()

Where the arguments mean the following:

- `channel`: A string describing the context. For example `"smtrat.parser"`.
- `msg`: The actual message as an expression that can be sent to a std::stringstream. For example `"foo: " << foo`.
- `args`: A description of the function arguments as an expression like `msg`.
- `condition`: A boolean expression that can be passed to `assert()`.

Typically, logging looks like this:

```
bool checkStuff(Object o, bool flag) {
    SMTRAT_LOG_FUNC("smtrat", o << ", " << flag);
    bool result = o.property(flag);
    SMTRAT_LOG_TRACE("smtrat", "Result: " << result);
    return result;
}
```

Logging is enabled (or disabled) by the `LOGGING` macro in CMake.

8.5 Statistics and timing

8.5.0.1 Statistics SMT-RAT has the ability to collect statistics after the solving process is finished.

For enabling this feature, the `SMTRAT_DEVOPTION_Statistics` needs to be turned on in the CMake settings.

8.5.0.1.1 Collecting statistics A statistics object can be created by inheriting from `Statistics.h`:

```
#pragma once
#include <smtrat-common/statistics/Statistics.h>
#ifdef SMTRAT_DEVOPTION_Statistics
namespace smtrat {
namespace myModule {
class MyStatistics : public Statistics {
private:
    std::size_t mCounter = 0;
public:
    void collect() { // called after the solving process to collect statistics
        Statistics::addKeyValuePair("counter", mCounter);
    }
    void count() { // user defined
        ++mCounter;
    }
};
}
}
#endif
```

This is then instantiated by calling

```
#ifdef SMTRAT_DEVOPTION_Statistics
auto& myStatistics = statistics_get<myModule::MyStatistics>("MyModuleName");
#endif
```

or, as shortcut

```
SMTRAT_STATISTICS_INIT(myModule::MyStatistics, myStatistics, "MyModuleName")
```

and statistics can be collected by calling the user defined operations (e.g. `myStatistics.count()`).

All statistics-related code should be encapsulated by the `SMTRAT_DEVOPTION_Statistics` flag. Alternatively, code can be encapsulated in `SMTRAT_STATISTICS_CALL()`.

8.5.0.2 Timing The statistics framework has the ability to easily collect timings.

The following code will measure the total running time of the code block as well as the number of times the code block was executed:

```
class MyStatistics : public Statistics {
private:
    carl::statistics::timer mTimer;

public:
    void collect() {
        Statistics::addKeyValuePair("timer", mTimer);
    }
    auto& timer() {
        return mTimer;
    }
};

SMTRAT_STATISTICS_INIT(myModule::MyStatistics, myStatistics, "MyModuleName")

auto start = SMTRAT_TIME_START();
// expensive code
SMTRAT_TIME_FINISH(myStatistics.timer(), start);
```

The measured timings will then appear alongside the statistics.

Note that the resolution of these measurements is in 1 millisecond. Thus, be careful when interpreting results; especially, it measured code blocks should be big enough.

8.6 Testing

SMT-RAT and CArL use `GoogleTest` for unit testing. To do so, go to the `src/tests/` folder and create a test analogously to the other tests (i.e. creating a new folder, adapting the `CMakeList.txt` in the new folder and in the `tests` folder). *Note that in the SMT-RAT repository are some obsolete tests.*

8.6.0.1 Utilities For creating CArL data structure in unit tests, we refer to `carl/src/util/parser/Parser.h`.

8.7 Validation

For debugging purposes, it can be useful to verify intermediate results (explanations, lemmas, etc) using an external SMT solver. SMT-RAT allows to store formulas during the solving process which are written to a `smt2` file once the solving process is finished.

8.7.0.1 Enabling this feature For enabling this feature, the `SMTRAT_DEVOPTION_Validation` needs to be turned on in the CMake settings.

8.7.0.2 Logging formulas The API allows to create a *validation point* with a given channel and a name. The channel and name should identify the validation point uniquely. The channel (e.g. `smtrat.modules.vs`) can be used to turn on and off validation points (similarly to logging channels) while the name (e.g. `substitution_result`) further distinguishes validation points within a channel.

To initialize a validation point with channel and name and sotre its reference to a member, use

```
SMTRAT_VALIDATION_INIT(channel, name, member);
```

Hint: to put it in a static member, use

```
SMTRAT_VALIDATION_INIT_STATIC(channel, name, member);
```

To an initialized validation point stored in a member, we can add a formula to be assumed to be satisfiable (consistent = true) or unsatisfiable (consistent = false). Each formula added to a validation point gets a unique index (given incrementally), which is also logged in the given channel with debug level.

```
SMTRAT_VALIDATION_ADD_TO(member, formula, consistent);
```

To combine the two steps above, use:

```
SMTRAT_VALIDATION_ADD(channel, name, formula, consistent);
```

8.7.0.3 Command line usage By appending the command line parameter `--validation.export-smtlib`, the formulas are stored to an smtlib file (by default `validation.smt2`, can be customized using `--validation.smtlib-filename`).

Note that all channels of interest need to be activated explicitly with `--validation.channel channel1 --validation.channel channel2 ...`. Furthermore, `--validation.channel path.to` will activate all channels starting with `path.to`, i.e. `path.to.channel1`, `path.to.channel2` etc.

8.8 Checkpoints

Checkpoints are useful for debugging a run of SMT-RAT, i.e. if an implementation of SMT-RAT behaves the same as some other implementation or some pseudocode.

To do so, we can insert checkpoints with a channel, a name, and a list of arguments of arbitrary type into the code. During a run, a sequence of checkpoints is produced which represent a run of the algorithm. We then can specify test cases which set up a reference run and then start the solver on an according input. SMT-RAT will then check during this run whether all checkpoints are hit in the correct order and will give output on that.

8.8.0.1 Enabling this feature You need to turn on the CMake option `SMTRAT_DEVOPTION_Checkpoints`.

8.8.0.2 Specyfing checkpoints Insert

```
SMTRAT_CHECKPOINT("channel_name", "checkpoint_name", parameter1, parameter2, ...);
```

into the code.

8.8.0.3 Setting up a reference run Create a test case in `src/test`. Include `#include <smtrat-common/smtrat-com h>` and add checkpoints using

```
SMTRAT_ADD_CHECKPOINT("channel_name", "checkpoint_name", false, parameter1, parameter2, ...)
```

or

```
SMTRAT_ADD_CHECKPOINT("channel_name", "checkpoint_name", true, parameter1, parameter2, ...)
```

to assert that a checkpoint is reached.

Afterwards, run the respective SMT-RAT code.

You can clear all checkpoints of a channel using

```
SMTRAT_CLEAR_CHECKPOINT("channel_name");
```

to reset a channel, that is, removing all checkpoints and resetting the pointer.

8.9 Finding and Reporting Bugs

This page is meant as a guide for the case that you find a bug or any unexpected behaviour. We consider any of the following events a (potential) bug:

- SMT-RAT crashes.
- A library used through SMT-RAT crashes.
- SMT-RAT gives incorrect results.
- SMT-RAT does not terminate (for reasonably sized inputs).
- SMT-RAT does not provide a method or functionality that should be available according to this documentation.
- SMT-RAT does not provide a method or functionality that you consider crucial or trivial for some of the datastructures.
- Compiling SMT-RAT fails.
- Compiling your code using SMT-RAT fails and you are pretty sure that you use SMT-RAT according to this documentation.

In any of the above cases, make sure that:

- You have installed all necessary dependencies in the required versions.
- You work on something that is similar to a system listed as supported platform at getting_started.
- You can (somewhat reliably) reproduce the error with a (somewhat) clean build of SMT-RAT. (i.e., you did not screw up the CMake flags, see build_cmake for more information)
- You compile either with `CMAKE_BUILD_TYPE=DEBUG` or `DEVELOPER=ON`. This will give additional warnings during compilation and enable assertions during runtime. This will slow down SMT-RAT significantly, but detect errors before an actual crash happens and give a meaningful error message in many cases.

If you are unable to solve issue yourself or you find the issue to be an actual bug in SMT-RAT, please do not hesitate to contact us. You can either contact us via email (if you suspect a configuration or usage issue on your side) or create a ticket in our bug tracker (if you suspect an error that is to be fixed by us). The bug tracker is available at https://github.com/smtrat/smtrat/issues.

When sending us a mail or creating a ticket, please provide us with:

- Your system specifications, including versions of compilers and libraries listed in the dependencies.
- The SMT-RAT version (release version of git commit id).
- A minimal working example.
- A description of what you would expect to happen.
- A description of what actually happens.

9 NewCoveringModule #and

Author: Philip Kroll (Philip.Kroll@rwth-aachen.de)

9.1 Introduction

New implementation of the CDCAC algorithm, also called the covering algorithm. The basic implementation is based on the paper Deciding the Consistency of Non-Linear Real Arithmetic Constraints with a Conflict Driven Search Using Cylindrical Algebraic Coverings. In short↩ : The algorithm is a variant of Cylindrical Algebraic Decomposition (CAD) adapted for satisfiability, where solution candidates (sample points) are constructed incrementally, either until a satisfying sample is found or sufficient samples have been sampled to conclude unsatisfiability. The choice of samples is guided by the input constraints and previous conflicts.

9.2 Usage

In this implementation as much code as possible from src/smtrat-cadcells. This also includes the projection operators implemented in src/smtrat-cadcells/operators and the representation heuristics implemented in src/smtrat-cadcells/representation. We also reuse the code from src/smtrat-mcsat/variableordering/VariableOrdering.h to calculate the used variable ordering. Change of the strategy to calculate the variable ordering, the used heuristics or the used projection operator, can be done by changing the respective settings in src/smtrat-modules/New↩ CoveringModule/NewCoveringSettings.h

9.3 Efficiency

The implementation also supports backtracking and incrementality, both of which are not covered in detail by the paper. The following cases are possible depending on what the previous result of the Covering Algorithm was :

- Previous result was SAT, i.e. a satisfying assignment was found and the cached coverings for all dimensions are partial
  - addConstraintSAT() is called with the new constraint and these are checked for satisfiability. The lowest level with an unsatisfied new constraints is returned. This also means that when all constraints SAT is concluded and no calculations are done.
  - removeConstraintSAT() is called with constraints that have to be removed from the cached partial coverings for all dimensions. This only makes the covering smaller, and the assignment is still satisfying.
- Previous result was UNSAT, i.e. the stored covering for level 0 is complete and thus no satisfying assignment could be found.
  - addConstraintUNSAT() is called with the new constraint. As the covering is unsatisfiable anyways these constraints are just added without further calculations.
  - removeConstraintUNSAT() is called with constraints that have to be removed from the cached complete coverings all dimensions.

References

[1] Cristina Borralleras, Salvador Lucas, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Sat modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, Jan 2012.

[2] Martin Bromberger and Christoph Weidenbach. Fast cube tests for lia constraint solving. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning*, pages 116–132, Cham, 2016. Springer International Publishing.

[3] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Satisfiability modulo transcendental functions via incremental linearization. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 95–113, Cham, 2017. Springer International Publishing.

[4] F. Corzilius and E. Ábrahám. Virtual substitution for SMT solving. In 18th Int. Symp. on Fundamentals of Computation Theory, 2011.

[5] 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.

[6] 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.

[7] Pascal Fontaine, Mizuhito Ogawa, Thomas Sturm, and Xuan Tung Vu. Subtropical satisfiability. In *International Symposium on Frontiers of Combining Systems*, pages 189–206. Springer, 2017.

[8] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. Sat solving for termination analysis with polynomial interpretations. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 340–354, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[9] 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.

[10] Alberto Griggio. A practical approach to satisfiability modulo linear integer arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:1–27, 2012.

[11] Marta Grobelna. Solving pseudo-boolean constraints, 2017.

[12] Stefan Herbort and Dietmar Ratz. Improving the efficiency of a nonlinear-system-solver using a componentwise newton method. 1997.

[13] Ahmed Irfan. *Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions*. PhD thesis, University of Trento, 2018.

[14] 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.

[15] Gereon Kremer, Florian Corzilius, and Erika Ábrahám. A Generalised Branch-and-Bound Approach and Its Application in SAT Modulo Nonlinear Integer Arithmetic. In *Proc. of CASC'16*, pages 315–335. Springer, 2016.

[16] Gereon Kremer and Erika Ábrahám. Fully incremental cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 2019.

[17] Andreas Krüger. Bitvectors in smt-rat and their application to integer arithmetics, 2015.

[18] 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.

[19] 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.

[20] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557 – 580, 2007. Special Issue: 16th International Conference on Rewriting Techniques and Applications.

[21] Aklima Zaman. Incremental linearization for sat modulo real arithmetic solving, 2019.

[22] Ömer Sali. Linearization techniques for nonlinear arithmetic problems in smt, 2018.