SMT-RAT

Version 0.4.0

Satisfiability-Modulo-Theories Real Arithmetic Toolbox

Manual

Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám

Contents

1 Introduction **3**

2 Installation **5**
2.1 Requirements 5
2.2 Building, installing and uninstalling `SMT-RAT` 5

3 General framework of `SMT-RAT` **7**
3.1 The `Module` class. 8
3.2 The `Manager` and the `Strategy` class 8

4 Embedding of a theory solver into an SMT solver **11**
4.1 Interfaces of a theory solver composed with `SMT-RAT` 11

5 Implementing further modules **13**
5.1 Quickstart 13
5.2 Members of a modul 14
5.3 Virtual interfaces 15
5.3.1 Informing about a constraint 16
5.3.2 Adding a constraint 16
5.3.3 Checking for consistency 16
5.3.4 Pushing a backtrack point 17
5.3.5 Popping a backtrack point 17
5.4 Running the backend module 18

6 Composing a solver **21**
6.1 The strategy 21
6.2 Specifying a strategy with the GUI 22

1 Introduction

This manual describes `SMT-RAT`, a toolbox which is specifically tailored to be used by an SMT solver in order to solve non-linear real arithmetic (NRA) efficiently. NRA is an important but hard-to-solve theory and only a fragment of it can be handled by some of the currently existing SMT solvers. The toolbox consists of four modules, implementing the virtual substitution method, the cylindrical algebraic decomposition method, a groebner bases simplifier and a general simplifier, respectively. The intention of the toolbox is that an SMT-compliant theory solver can be achieved by composing these modules according to a user defined strategy and with the goal to exploit their advantages. Furthermore, it supports the addition of further modules implementing other methods dealing with real arithmetic.

2 Installation

2.1 Requirements

SMT-RAT has been successfully compiled and tested under Linux. For its configuration we use cmake, which can be found on http://www.cmake.org/, and for its compilation we use g++ (version 4.7 or higher), available on http://gcc.gnu.org/. The configuration settings can be changed with the command line interface ccmake. The arithmetic calculations within SMT-RAT use the C++ library GiNaC, which can be found on http://www.ginac.de/. The algebraic operations and data structures within SMT-RAT are based on the C++ library GiNaCRA available at http://ginacra.sourceforge.net/. Optional, there is a grapical user interface for the composition of a solver, which needs java (version 1.7 or higher). Note, that we use C++11. Summarizing, you need the following packages before installation:

- g++ (version 4.7 or higher)
- cmake
- ccmake
- GiNaC
- GiNaCRA
- java (version 1.7 or higher)

2.2 Building, installing and uninstalling SMT-RAT

You can download SMT-RAT from http://smtrat.sourceforge.net/ and install it the following way:

1. Unpack the package:

   ```
   tar xvzf smtrat-*.tar.gz
   ```

2. Create a directory (e.g. build) that will contain the object files and the executables, and change into it:

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

3. Configure:

```
cmake ..
```

4. Build:

```
make
```

5. Install:

```
make install
```

6. Uninstall:

```
xargs rm < install_manifest.txt
```

```
make clean
```

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

3 General framework of SMT-RAT

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

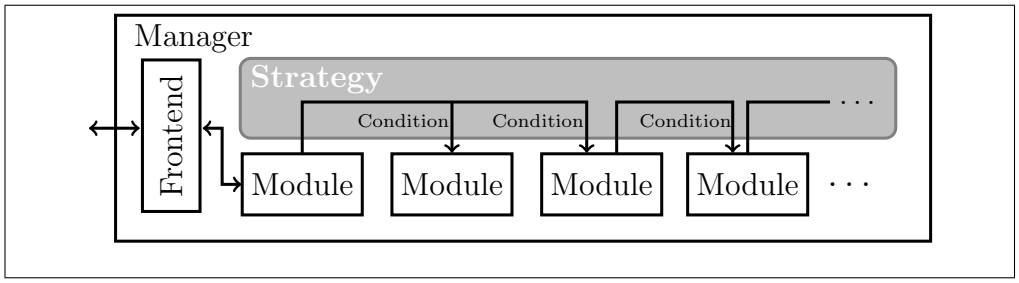

Figure 3.1: A snapshot of an SMT-RAT composition being an SMT solver for NRA.

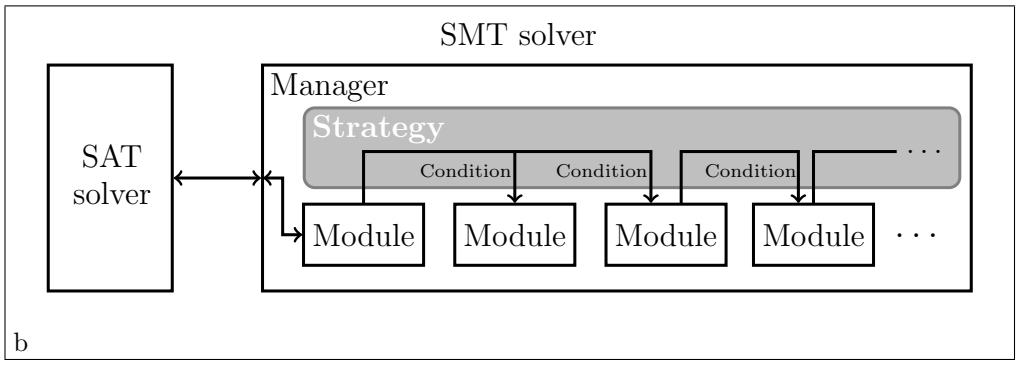

Figure 3.2: A snapshot of an SMT-RAT composition being a theory solver embedded in an SMT solver.

SMT-RAT defines three types of components: the *manager*, the *strategy* and, as already mentioned, *modules*. In addition, a frontend (1) provides the interfaces to an extern SMT solver or (2) parses an input file to a non-linear real arithmetic formula, we denote in the following just as formula. In this section we first describe the functionality of a module and, then, show how the manager composes different modules according to a strategy to a solver.

3.1 The `Module` class.

A module m contains a set of formulas, called its *set of received formulas* and denoted by $C_{rcv}(m)$. The main procedure of a module is `isConsistent()` and either decides whether $C_{rcv}(m)$ is satisfiable or not returning `sat` or `unsat`, respectively, or returns `unknown`. Note, that a set of formulas is semantically defined by their conjunction. We can manipulate the set of received formulas by adding (removing) formulas φ to (from) it with `add(`φ`)` (`remove(`φ`)`). Usually, $C_{rcv}(m)$ is only changed slightly between two consecutive `isConsistent()` calls, hence, the solver's performance can be significantly improved if the modules can make use of the results of previous checks, that is they support *incrementality* and *backtracking*. In case that the module determines the unsatisfiability of $C_{rcv}(m)$, it is expected to compute at least one preferably small *infeasible subset* $C_{inf}(m) \subseteq C_{rcv}(m)$. Moreover, a module has the possibility to name lemmas, which are formulas being tautologies, that is they hold for all assignments of the variables occuring in them. These lemmas should encapsulate information which can be extracted from a module's internal state and propagated among other SMT-RAT modules. Furthermore, SMT-RAT provides the feature that a module itself can ask other modules for the satisfiability of a set of formulas, called its *set of passed formulas* denoted by $C_{pas}(m)$, using the procedure `runBackends()` which is controlled by the manager.

3.2 The `Manager` and the `Strategy` class

SMT-RAT allows a user to decide how to compose the different procedures, which are implemented in the modules. For this purpose we provide a GUI where the user can create such a composition we call strategy. Further explanation can be found in Chapter 6. Here, it is enough to know that a *strategy* is a directed tree $T := (V, E)$ with a set V of (instances of) modules as nodes and $E \subseteq V \times \Omega \times V$, where Ω is a set of conditions. The *manager* contains the strategy and the input formula C_{input}, either received by a prefexed solver or parsed from an example file. Furthermore, it maintains the allocation of modules as follows.

Initially, the manager calls the method `isConsistent()` of the module m_r given by the root of the strategy with $C_{rcv}(m_r) = C_{input}$ being the set of received formulas of this module. Whenever a module $m \in V$ calls `runBackends()`, with $C_{pas}(m)$ being its set of passed formulas, the manager calls `isConsistent()` of each module m' with C_{rcv} (m') = C_{pas} (m) being its set of received formulas, for which an edge $(m, \omega, m') \in E$

exists such that ω holds for $C_{pas}(m)$, and passes the results back to m. Furthermore, it also passes back the infeasible subsets and lemmas provided by the invoked modules. The module m can now benefit in its solving and reasoning process from this shared information.

A condition $\omega \in \Omega$ on real arithmetic formulas is an arbitrary Boolean combinations of properties of a real arithmetic formula such as..

- .. propositions about the Boolean structure of the formula, e.g., whether it is a conjunction of real arithmetic constraints or whether it is in conjunctive normal form.
- .. propositions about the constraints in the formula, e.g. whether the formula contains (does not contain) (only) equations/(strict inequalities).
- .. propositions about the polynomials compared by the constraints, e.g., whether they are linear, univariate, contain more than n variables or have a degree smaller than k.

Here is a small example of a strategy forming a complete SMT solver for NRA. We write short (m, m') for (m, ω, m) if $\omega =$ `true`, which is the condition which holds for all NRA formulas. We construct the simple strategy defined by the nodes $\mathtt{M}_{\mathtt{CNF}}$, $\mathtt{M}_{\mathtt{SAT}}$ and $\mathtt{M}_{\mathtt{CAD}}$ and the edges $(\mathtt{M}_{\mathtt{CNF}}, \mathtt{M}_{\mathtt{SAT}})$ and $(\mathtt{M}_{\mathtt{SAT}}, \mathtt{M}_{\mathtt{CAD}})$. The root module $\mathtt{M}_{\mathtt{CNF}}$ transforms its set of received formulas $C_{rcv}(\mathtt{M}_{\mathtt{CNF}})$ to an equisatisfiable set of clauses $C_{pas}(\mathtt{M}_{\mathtt{CNF}})$ and calls `runBackends()`. The invoked backend is a SAT-solver module $\mathtt{M}_{\mathtt{SAT}}$, which runs DPLL-style SAT-solving on the Boolean abstraction of the set of received clauses $C_{rcv}(\mathtt{M}_{\mathtt{SAT}}) = C_{pas}(\mathtt{M}_{\mathtt{CNF}})$. $\mathtt{M}_{\mathtt{SAT}}$ might call `runBackends()` for a partial Boolean assignment corresponding to a set of NRA constraints, which form $C_{pas}(\mathtt{M}_{\mathtt{SAT}})$, in order to check for its consistency. In literatur such a backend call is often refered to as *theory call*. The backend module $\mathtt{M}_{\mathtt{CAD}}$ determines whether this set of constraints is consistent and provides infeasible subsets if not. Furthermore, it could provide lemmas. In both cases $\mathtt{M}_{\mathtt{SAT}}$ abstracts the according NRA formulas to a Boolean formula and stores them as learned clauses.

Infeasible subsets and lemmas, which contain only formulas from $C_{pas}(\mathtt{M}_{\mathtt{SAT}})$, prune the 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. Lemmas containing new constraints we call *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 can not be guaranteed.

4 Embedding of a theory solver into an SMT solver

In this section we show how to embed a composed theory solver according to Chapter 6, e.g., the built-in theory solver `NRATSolver`, into an SMT solver in order to extend its supported logics by NRA. Firstly, we will explain how to create an NRA formula and, afterwards, we will the introduce the interfaces of the class `Manager`, which form the interfaces of a theory solver composed by `SMT-RAT`.

4.1 Interfaces of a theory solver composed with `SMT-RAT`

An SMT solver can use the following interfaces:

- `bool inform( const string& _constraint, bool _infix )`

  Informs the theory solver about the existence of the given constraint in form of a string. The second argument is a flag, which indicates whether the given constraint is written in infix or prefix notation.

- ```
  bool addConstraint
  (
      const string& _constraint,
      const bool _infix,
      const bool _polarity
  )
  ```

  Adds the given constraint in form of a string to the theory solver. The second argument is a flag, which indicates whether the given constraint is written in infix or prefix notation. The last argument is again a flag which is `true` if the constraint has to hold in the given form and `false` if its inversion has to hold. The inversion operator for a constraint simply changes its relation symbol in the following way:

  $$\begin{array}{ccc} = & \mapsto & \neq \\ \neq & \mapsto & = \\ \leq & \mapsto & > \\ \geq & \mapsto & < \\ < & \mapsto & \geq \\ > & \mapsto & \leq \end{array}$$
  ```

- `Answer isConsistent( )`

  Checks the so far received constraints for consistency. The answer can either be `true`, if the set of the so far received constraints is consistent, or `false`, if it is inconsistent, and `unknown`, if the theory solver cannot reason about it.

- `void pushBacktrackPoint( )`

  Pushes a backtrack point to the stack of backtrack points.

- `void popBacktrackPoint( )`

  Pops a backtrack point from the stack of backtrack points and undoes everything which has been done after adding that backtrack point.

- `vector< vector< unsigned > > getReasons( )`

  Returns one or more reasons for the inconsistency of the constraints. A reason is an infeasible subset of the so far received constraints. An element of a reason is a number i and means that the ith received constraint is an element of this infeasible subset.

5 Implementing further modules

In this chapter we explain how to implement further modules. One principle of the design of SMT-RAT is that you can take advantage of all the features it provides, but you do not have to.

5.1 Quickstart

Figure 5.1 shows the most rudimentary way of how a module can be implemented. Furthermore, there are a few more things which must be added to SMT-RAT:

1. Add the module type MT_MyModule to the enum ModuleType of all module types in src/ModuleType.h.

2. The file src/modules/Modules.h has to include the header of the new module, i.e. add "#include "MyModule.h"" to modules/Modules.h.

3. Extend the mapping of module types to module factories in the constructor of the class Manager:

```
addModuleType
(
    MT_MyModule,
    new StandardModuleFactory<MyModule>()
);
```

4. Add a proposition to the class src/Formula.h which expresses for a formula that your module cannot solve it, that is returns unknown or calls its backend (for further details to this see Section 5.4). Each existing proposition is represented by a bit in a bitset within Formula, currently of size 64. Hence, the proposition to add has to take an unused bit. If all bits are used the bitset must be extended. An example of how to add this proposition is given by Figure 5.2.

5. Extend the method Formula::notSolvableBy by the case that _moduleType is MT_MyModule returning the before created proposition as shown in Figure 5.3

6. Extend the method Formula::resetSolvableByModuleFlags...

7. Finally, the file src/CMakeLists.txt has to be updated. Add to the set of lib_modules_headers "src/modules/MyModule.h" and to the set of lib_modules_src "src/modules/MyModule.cpp".

```
#include "../Module.h"

class MyModule : public Module
{
public:
    MyModule( Manager* const _manager )
        : Module( _Manager )
    { mModuleType = MT_MyModule; }

    ~MyModule( ) { }
};
```

Figure 5.1: The most rudimentary way of a module can be implemented.

```
..
    static const Condition PROP_CANNOT_BE_SOLVED_BY_MYMODULE
        = Condition().set( 53, 1 );
..
```

Figure 5.2: Example of how to add a proposition to `Formula.h`.

The generated module now uses the straightforward implementation of the superclass `Module`. This includes many useful methods to read and manipulate the data in the module and the interfaces which can be extended such that they contain the actual implementation of your module.

5.2 Members of a modul

Before we explain the virtual interfaces containing the actual implementation of your modul we give a small description of the members of `Module`.

- `vector< unsigned > mBackTrackPoints`

  Stores the backtrack points.

- `vector< Formula > mInfeasibleSubsets`

  Stores the infeasible subsets of the so far received constraints, if the module determined their inconsistency before. An object of the type `Formula` currently consists of a conjunction of constraints and stores which of the propositions, defined in Chapter 6, hold.

- `Manager* const mpManager`

  A pointer to the manager which stores this module. By this pointer the modul can obtain its backend, which gets assigned by the manager as it implements the strategy.

```
void Formula::notSolvableBy( ModuleType _moduleType )
{
    switch( _moduleType )
    {
    ..
    case MT_MyModule:
    {
        mPropositions |= PROP_CANNOT_BE_SOLVED_BY_MYMODULE;
        break;
    }
    ..
    }
}
```

Figure 5.3: How to extend the method `Formula::notSolvableBy` in `Formula.cpp`.

- `ModuleType mModuleType`

  The type of this module.

- `vector< Module* > mBackend`

  The pointer to the module's backend. The backend is stored in the manager.

- `vector< unsigned > mBackendBacktrackpoints`

  The backtrack points in the backend.

- `bool mBackendUptodate`

  A flag which indicates that the current backend considers the so far passed constraints.

- `Formula* mpReceivedConstraints`

  The vector of the so far received constraints.

- `Formula* mpPassedConstraints`

  The vector of the so far passed constraints.

5.3 Virtual interfaces

In the following we explain the virtual interfaces of the module, showing what their implementation in `Module` already provides and how they should be extended in your module.

5.3.1 Informing about a constraint

`void inform( Constraint* _constraint )`

Informs the module about the existance of the given constraint. This information might be useful for the module. The implementation in the `Module` does just nothing by calling this method.

```
void MyModule::inform( Constraint* _constraint )
{
    /*
     * Write the implementation here.
     */
}
```

Figure 5.4: Example showing how to implement the method `inform`

5.3.2 Adding a constraint

`bool addConstraint( Constraint* _constraint )`

Adds a constraint to the vector of the so far received constraints of the module. If the module can already decide whether the given constraint is not consistent, it returns `false` otherwise `true`. This is exactly how it is implemented in `Module`, unless it does no kind of consistency check on the given constraint, that is it always adds the given constraint to the vector of the so far received constraints and returns `true`. Note, that implementing your own module might need some initialization in this method, but you should always call `Module::addConstraint` at the beginning of your implementation of `addConstraint`.

```
bool MyModule::addConstraint( Constraint* _constraint )
{
    Module::addConstraint( _constraint );
    /*
     * Write the implementation here.
     */
}
```

Figure 5.5: Example showing how to implement the method `addConstraint`

5.3.3 Checking for consistency

`Answer isConsistent()`

Implements the actual consistency check of the constraints this module has so far received. There are three options how this module can answer: it can determine that these constraints are consistent and returns `true`, or it determines inconsistency and returns `false`. Otherwise, it returns `unknown`. If this module has a backend, it can call it to check the consistency of any set of constraints. How to run the backend is shown in Section 5.4. The method `Module::isConsistent` does nothing more than directly calling its backend on the so far received constraints. A module has also the opportunity to reason about the conflicts occured, if it determines inconsistency. For this purpose it can store several infeasible subsets of the set of so far received constraints, which can be accessed as seen before.

```
Answer MyModule::isConsistent( )
{
    /*
     * Write the implementation here.
     */
}
```

Figure 5.6: Example showing how to implement the method `isConsistent`

5.3.4 Pushing a backtrack point

`void pushBacktrackPoint()`

Adds a backtrack point to the stack of backtrackpoints. This is already implemented by `Module::pushBacktrackPoint`. It might be necessary to store the current state of the datastructure used in your module.

```
void MyModule::pushBacktrackPoint( )
{
    Module::pushBacktrackPoint();
    /*
     * Write the implementation here.
     */
}
```

Figure 5.7: Example showing how to implement the method `pushBacktrackPoint`

5.3.5 Popping a backtrack point

`void popBacktrackPoint()`

Removes the last backtrack point from the stack of backtrack points and undoes everything which has been done after adding that backtrack point.

`Module::popBacktrackPoint` already removes the corresponding constraints from the vector of the so far received constraints, applies `popBacktrackPoint` in its backend until it does not contain any constraint created as a consequence of a removed received constraint, and removes all infeasible subsets containing removed received constraints. If you implement `popBacktrackPoint` make sure that everything stored additionally, which depends on the removed received constraints, gets deleted. Call `Module::popBacktrackPoint` afterwards.

```
void MyModule::popBacktrackPoint( )
{
    /*
     * Write the implementation here.
     */
    Module::popBacktrackPoint();
}
```

Figure 5.8: Example showing how to implement the method `popBacktrackPoint`

5.4 Running the backend module

Fortunately, there is no need to manage the addition of constraints to the backend, or pushing and popping its backtrack points. This would be even more involved as we do allow changing the backend if it is appropiate (more details to this are explained in Chapter 6). Running the backend is done in two steps:

1. Fill up the vector of the so far passed constraints.
2. Call `runBackend()`.

The first step is a bit more tricky, as we need to know which received constraints led to a passed constraint. For this purpose a modul contains a mapping from a passed constraint to one or more sets of received constraints. We give a small example by showing the behavior of one of the modules we provide, the `SimplifierModule`. Let us assume that this modul has so far received the following constraints:

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

`SimplifierModule` combines the first two constraints c_0 and c_1 to $c_3 : x = 0$. Then it combines c_3 with c_2 to $c_5 : x = 0$. Afterwards it calls its backend on the only remaining constraint, that is the passed constraints contain only $c_5 : x = 0$. The mapping of c_5 to the received constraints it stems from is

$$c_5 \ \mapsto\ (\{c_0,\ c_1\},\ \{c_2\}).$$

The mapping is maintained automatically and offers two methods to add constraints to the vector of the passed constraints:

- `bool addReceivedConstraintToPassedConstraints( unsigned _pos )`

  Adds the constraint at the given posititon of the vector of the so far received constraints to the vector of the passed constraints. The mapping to its *original constraints* contains only the constraint at the given position in the vector of received constraints. It returns `true` if a constraint has been added, which only occurs if the given position is indeed within the range of the vector of received constraints and this constraint has not yet been added.

- ```
  bool addConstraintToPassedConstraints
  (
      Constraint*                  _constraint,
      vector< set< Constraint* > > _origins
  )
  ```

  Adds the given constraint to the vector of the passed constraints. It is mapped to the given original constraints `_origins`. It returns `true` if a constraint has been added, which only occurs if the given constraint has not yet been added. Note, that `_origins` must contain only members of the vector of so far received constraints. We do not check this by reason of efficiency.
  ```

6 Composing a solver

`SMT-RAT` contributes a toolbox for composing an SMT compliant solver for real arithmetic, that means it is incremental, supports backtracking and provides reasons for inconsistency. The resulting solver is either a fully operative SMT solver for real arithmetic, 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 real arithmetic.

We are talking about composition and toolbox, as `SMT-RAT` contains implementations of many different mathematical approaches to tackle real arithmetic, each of them embedded in a module with uniform interfaces. Theses modules form the tools in the toolbox and it is dedicated to a user how we use them for solving a real arithmetic formula. We provide a self-explanatory graphical user interface (GUI) for defining a graph-based *strategy* specifying which module(s) should be applied on which formula, taking into account the modules we have already involved.

In the following of this chapter we firstly give a formal definition of a strategy, followed by a brief introduction to the existing modules equipped with an estimation of their input-based performances. Afterwards we illustrate how to use the GUI for composing a strategy.

6.1 The strategy

For the comprehension of this section we presume the reader to be familiar with the concept of `runBackends(..)`, which we introduced in Chapter **??**.

From the viewpoint of a user, who composes a solver with `SMT-RAT`, the composition starts solving a real arithmetic formula φ by invoking a consistency check of a module M on φ. Most of the modules simplify φ to an equisatisfiable real arithmetic formula φ' or need to know the consistency of a real arithmetic subproblem φ', which it cannot solve itself. In that case, M calls `runBackends(..)`, which runs the consistency check of a backend module M' on φ'. The strategy specifies M' according to, firstly, the structure of φ' and, secondly, the call hierarchy of the consistency checks of modules leading to this request.

Therefore, the strategy is a directed tree $T := (V, E)$ with a set V of module instances as nodes and $E \subseteq V \times \Omega \times V$, where Ω is a set of conditions on real arithmetic formulas. Initially, the method `isConsistent(`φ_{rcv}`)` of the module instance given by the root of the strategy is called, where φ_{rcv} is a conjunction of real arithmetic formulas. Whenever a module instance $m \in V$ calls `runBackends(`φ_{pass}`)`, `isConsistent(`φ_{pass}`)` of each module m' is called, for which an edge $(m, \omega, m') \in E$ exists such that ω holds for φ_{pass}, and passes the results back to m. A condition on real arithmetic formulas are arbitrary Boolean combinations of properties of a real arithmetic formula as follows.

- Propositions about the Boolean structure of the formula. In our case it is always a conjunction of positive literals, e.g. whether it is a conjunction of real arithmetic constraints only or whether it is in conjunctive normal form.
- Propositions about the constraints in the formula, e.g. whether the formula contains (does not contain) (only) equations/(strict inequalities).
- Propositions about the polynomials compared by the constraints, e.g., whether they are linear, univariate, contain more than n variables or have a degree smaller than k.

6.2 Specifying a strategy with the GUI