SMT-RAT
Version 0.0.2

Satisfiability-Modulo-Theories Real Arithmetic Toolbox

Manual

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

Contents

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++, available on http://gcc.gnu.org/. 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/. Summarizing, you need the following packages before installation:

- g++
- cmake
- GiNaC
- GiNaCRA

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

Our toolbox has a modular C++ class design which can be used to compose NRA theory solvers for an SMT embedding in a *dynamic* and *hierarchic* fashion. Our NRA theory solvers are instances of `Manager`, which offers an interface to communicate with the environment and which coordinates the satisfiability check according to a user-defined `Strategy`. Such a strategy combines basic NRA theory solver modules, derived from `Module`. Figure 3.1 shows an example configuration. Moreover, a Java-based graphical user interface (GUI) can be used for an intuitive and user-friendly specification of strategies (and the automatic generation of a corresponding `Strategy` class). Next, we briefly describe these concepts.

3.1 The `Formula` class

`Formula` instances contain, besides a sequence of NRA constraints, a bitvector storing some information about the problem and the history of its check. E.g., there is a bit which is 1 if some of the constraints are equations. Also for each module there is a bit which is 1 if the module was already invoked on the given problem. Such information can be used to specify conditions under which a procedure should be invoked for a certain problem.

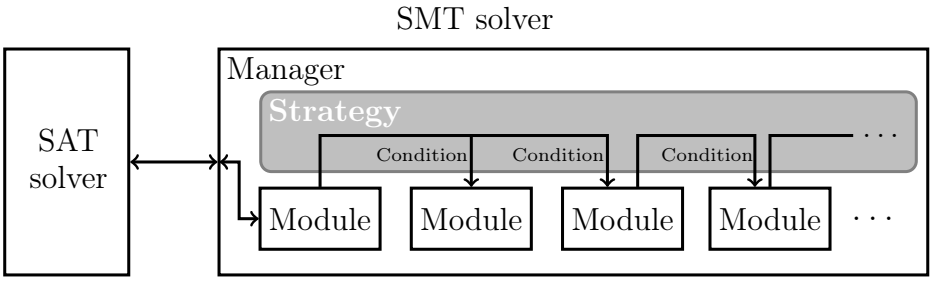
3.2 The `Module` class

A *module* is an SMT-compliant implementation of a procedure (e.g., constraint simplifier, an incomplete procedure or a complete decision procedure) which can be used for the satisfiability check of NRA formulas. A module's interface allows to add constraints, to push and pop backtrack points, to check the so far added constraints for consistency and to obtain an infeasible subset of these constraints if they are detected to be inconsistent.

Modules have the opportunity to call other modules (*backends*) on sub-problems. A novel achievement of our toolbox is that this call hierarchy is *dynamic* and guided by a *user-defined strategy*. Currently, we only support sequential execution, parallel solving is planned for later releases.

Inheritance can be used to extend existing modules. Besides the basic type `Module`, our toolbox offers five sub-classes. `SimplifierModule` ($\texttt{SI}_M$) implements smart simplifications, whereas `GroebnerModule` ($\texttt{GS}_M$) simplifies equation systems using Gröbner bases and probably detects inconsistency. The cylindrical algebraic decomposition method is implemented in `UnivariateCADModule` ($\texttt{UC}_M$) for the univariate case and in

Figure 3.1: A snapshot of an SMT-RAT composition embedded in an SMT solver.



`CADModule` ($\texttt{MC}_M$) for the general multivariate case. The last module class `VSModule` ($\texttt{VS}_M$) implements a version of the virtual substitution method.

3.3 The `Strategy` class

`SMT-RAT` offers a framework to integrate single modules to powerful and efficient *composed theory solvers*. The composition relies on a user-defined `Strategy` that specifies for each `Formula` instance which module should be used for its check. A strategy is basically a sequence of condition-module pairs. For each `Formula` instance, it determines the first module whose condition evaluates to true on the bitvector of the formula. E.g., the strategy "c_1 ? (m_1) : (c_2 ? (m_2) : (m_3))" determines m_1 as module type for φ if the bitvector of φ fulfills the condition c_1. If φ does not fulfill c_1 but c_2, then an instance of m_2 is called, otherwise of m_3.

3.4 The `Manager` class

The `Manager` contains references to the available module instances and to the user-defined strategy. It manages, on the one hand, the creation and linking of the modules, and, on the other hand, the communication between them and the environment, e.g., the frontend of an SMT solver.

4 Embedding of a theory solver into an SMT solver

In the following we assume, that either a composed theory solver according to Chapter 6 or the composition we provide, i.e. the `NRATSolver`, is used. They only differ in the strategy they implement and provide the same interfaces. We give a short description of these interfaces and give an example by embedding it into `OpenSMT`.

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.

4.2 Example: Embedding a SMT-RAT composition into OpenSMT

OpenSMT is an open source SMT solver and supports the embedding of additional theory solver. A detailed instruction of how to extend OpenSMT by a theory solver can be found on http://verify.inf.usi.ch/opensmt. Unfortunately, OpenSMT does not yet support quantifier free nonlinear real arithmetic (QF_NRA). For the following we assume that it supports QF_NRA and create a theory solver using the interfaces OpenSMT provides.

```
class TSolver
{
    void   inform        ( Enode * );
    bool   assertLit     ( Enode * );
    bool   check         ( bool );

    void   pushBktPoint  ( );
    void   popBktPoint   ( );
    bool   belongsToT    ( Enode * );
    void   computeModel  ( );

    vector <  Enode *  >  & explanation;
    vector <  Enode *  >  & deductions;
    vector <  Enode *  >  & suggestions;
}
```

OpenSMT gives the following explanation:

> "inform is used to communicate the existence of a new T-atom to the T-solver. assertLit asserts a (previously informed) T-atom in the T-solver with the right polarity; it may also perform some cheap form of consistency check. check determines the T-satisfiability of the current set of asserted T-atoms. pushBktPoint and popBktPoint are used respectively to save and to restore the state of the T-solver, in order to cope with backtracking within the SAT-Solver. belongsToT is used to determine if a given T-atom belongs to the theory solved by the T-solver. Finally computeModel forces T-solver to save the model (if any) inside Enode's field.
>
> Three vectors, explanation, deductions, suggestions, are shared among the T-solvers, and they are used to simplify the communication of conflicts, T-atoms to be deduced and "suggested" T-atoms. Suggestions are atoms consistent with the current state of the T-solver, but that they cannot be directly deduced. Suggestions are used to perform decisions in the SAT-Solver." [1]

We derive a theory solver and extend it by the three additional members

- mpManager, being a pointer to the theory solver object,
- mReceivedEnodes, needed to reconstruct which enode corresponds to which constraint,
- mBackendsBacktrackpoints, to store the backtrack points.

[1] http://verify.inf.usi.ch/opensmt/build-your-solver

Figure 4.1 shows the resulting code for the header of the theory solver within `OpenSMT`. The constructor of the embedded theory solver is shown in Figure 4.2. Note, that instead of `NRATSolver` any other `SMT-RAT`-composition can be used. Figure 4.3 shows the method which informs the theory solver about the existence of a constraint. Here, we first get the string representation of the constraint, which is in prefix notation, and pass it to the `NRATSolver`. Note, that, for a reason we are not aware of, the return value has to be `lbool` and not, as mentioned on the webpage, `void`. The method `assertLit`, given by Figure 4.4, takes again the string representation of the given literal/constraint and adds it to the `NRATSolver`. The return value is `false` if the constraint is inconsistent and `true` otherwise. Note, that the polarity indicates whether the literal is positive or negative. A negativ fulfilled literal implies that the aforementioned inversion of the constraint has to hold. The implementation of the method `pushBacktrackPoint` shown by Figure 4.5 is straightforward.

Figure 4.6 shows the implementation of the method `popBacktrackPoint`. It first empties the explanation vector and then removes all stored enodes, which have been added after the backtrack point to remove. Finally, it pops the backtrack point within the theory solver used by `OpenSMT` and calls `popBacktrackPoint` of the `NRATSolver`. The check procedure given in Figure 4.7 calls `isConsistent` of the `NRATSolver`. The easiest case is if the return value is `true`. Then we know that the set of the so far added constraints is consistent and return `true`. If it determines inconsistency, that is `isConsistent` returns `false`, we extract one reason for the inconsistency and store the corresponding literals in the explanation vector. This is why we need to store the so far received literals. Finally, we return `false`. Note, that we might obtain more than one reason. However, `OpenSMT` does not support more than one reason and we discard the other reasons. The case of getting the answer `unknown` is unfortunately not supported by `OpenSMT`.

Figure 4.1: The header implementation of a theory solver used by OpenSMT.

```
#include "TSolver.h"
#include <smt-rat/smt-rat.h>

class NRASolver : public OrdinaryTSolver
{
public:
    NRASolver( const int
           , const char *
           , SMTConfig &
           , Egraph &
           , SStore &
           , vector< Enode * > &
           , vector< Enode * > &
           , vector< Enode * > & );

   ~NRASolver ( );

   lbool   inform              ( Enode * );
   bool    assertLit           ( Enode *, bool = false );
   void    pushBacktrackPoint  ( );
   void    popBacktrackPoint   ( );
   bool    check               ( bool );
   bool    belongsToT          ( Enode * );
   void    computeModel        ( );
#ifdef PRODUCE_PROOF
   Enode * getInterpolants     ( );
#endif

private:
    Manager*         mpManager     ;
    vector< Enode* >   mReceivedEnodes ;
    vector< unsigned > mBacktrackPoints;
};
```

Figure 4.2: The constructor of a theory solver used by OpenSMT.

```
NRASolver::NRASolver( const int           i
                    , const char *        n
                    , SMTConfig &         c
                    , Egraph &            e
                    , SStore &            t
                    , vector< Enode * > & x
                    , vector< Enode * > & d
                    , vector< Enode * > & s )
             : OrdinaryTSolver ( i, n, c, e, t, x, d, s )
{
    mpManager      = new NRATSolver()    ; \\ This could also be
                                           \\ your theory solver
                                           \\ composition.
    mReceivedEnodes  = vector< Enode* >()  ;
    mBacktrackPoints = vector< unsigned >();
}
```

Figure 4.3: The method to inform the theory solver used by OpenSMT about a constraint.

```
lbool NRASolver::inform(Enode *e)
{
    (void)e;
    assert(e);
    assert(belongsToT(e));

    ostringstream stream;
    stream << e;

    mpManager->inform( stream.str(), false );

    return l_Undef;
}
```

Figure 4.4: The method to assert a literal in a theory solver used by OpenSMT.

```
bool NRASolver::assertLit( Enode *e, bool reason )
{
    (void)e;
    (void)reason;
    assert(e);
    assert(belongsToT(e));

    mReceivedEnodes.push_back( e );

    ostringstream stream;
    stream << e;

    return mpManager->addConstraint( stream.str()            ,
                                     false                   ,
                                     e->getPolarity()==l_True );
}
```

Figure 4.5: The method to push a backtrack point in a theory solver used by OpenSMT.

```
void NRASolver::pushBacktrackPoint()
{
    mBacktrackPoints.push_back( mReceivedEnodes.size() );
    mpManager->pushBacktrackPoint();
}
```

Figure 4.6: The method to pop a backtrack point in a theory solver used by `OpenSMT`.

```
void NRASolver::popBacktrackPoint( )
{
    explanation.clear();

    while( mBacktrackPoints.back()<mReceivedEnodes.size() )
    {
        mReceivedEnodes.pop_back();
    }

    mBacktrackPoints.pop_back();

    mpManager->popBacktrackPoint();
}
```

Figure 4.7: The method to check for consistency in a theory solver used by `OpenSMT`.

```
bool NRASolver::check( bool _complete )
{
    (void)_complete;

    switch( mpManager->isConsistent() )
    {
        case TS_True: return true;
        case TS_False:
        {
            vector< vector< unsigned > > reasons
                = mpManager->getReasons();
            vector< unsigned >::const_iterator pos
                = reasons.back().begin();
            while( pos!=reasons.back().end() )
            {
                explanation.push_back( mReceivedEnodes.at( *pos ) );
                ++pos;
            }
           return false;
        }
        case TS_Unknown: assert( false );
        default: assert( false );
    }
}
```

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 5.1.

5. Extend the method Formula::notSolvableBy by the case that _moduleType is MT_MyModule returning the before created proposition as shown in Figure 5.1

6. 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".

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

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

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

    ~MyModule( ) { }
};
```

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

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

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.

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

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

- `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.

- `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.

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

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

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`.

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

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

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

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.

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

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

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.

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

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

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

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.

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

```
Answer MyModule::popBacktrackPoint( )
{
    /*
     * Write the implementation here.
     */
    Module::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 theory solver

First of all, SMT-RAT contributes an SMT compliant theory solver for full real arithmetic. That means it is incremental, supports backtracking and provides reasons for inconsistency. In addition, it allows a user to compose a theory solver with the given modules in SMT-RAT and any further implemented modules (see Chapter 5). This chapter shows how to create a theory solver implementing a user defined strategy.

6.1 Creating a theory solver

Creating a theory solver is very easy. Basically, it is a subclass of the class Manager and does not need to implement any function. Everything, except the definition of the strategy, is provided by the class Manager, that is all the interfaces required by an exterior SMT solver and all the managment which links the input formula, the module instances and the given strategy. Figure 6.1 shows an example of how to implement a theory solver, which does not use any module and hence returns unknown to all input instances.

Figure 6.1: The implementation of a theory solver without defining the strategy.

```
#include "Manager.h"

class YourTheorySolver: public Manager
{
    public:

    YourTheorySolver()
    {
     // Here you define later the strategy.
    };

    ~YourTheorySolver() {};
};
```

6.2 Implementing a strategy in a theory solver

This section gives the definition of a strategy and shows how it can be implemented in a theory solver. But before we come to these two points, we analyse the requirements to the strategy.

We want modules to be able to use another module as backend. The module calls its backend on a formula, which it cannot check for consistency by itself. Depending on this formula the strategy specifies the backend the module has to call. So, basically the strategy has to specify which is the module we run (as backend) on a given formula. Note, that we are going to consider calling more than one backend in parallel in the next release of `SMT-RAT`.

6.2.1 The strategy

Knowing the requirements to the strategy, we can now define its grammar. Let M be the set of all module types. Note, that we can use more than one instance of a module type in the resulting theory solver. Let $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ and *Prop* be a set of propositions about real arithmetic formulas. Let $Formula_{RA}$ be a set of pairs of a conjunction of real arithmetic constraints and a set of propositions $P \subseteq Prop$. Then, a strategy is defined by the abstract grammar

$$\text{strategy} \quad ::= \quad \text{c ? m : (strategy)} \quad | \quad \text{m}$$

where c is a Boolean combination of propositions in *Prop* and $m \in M$.

Let *Strat* be the set of all possible strategies. Then, the function

$$\alpha : \ Strat \times Formula_{RA} \ \rightarrow \ M$$

mapping a strategy and a formula to a module (representing the demanded backend) is defined as follows:

$$\begin{aligned} \alpha(m,\ (\varphi,\ P)) &= m \\ \alpha(c\ ?\ m : (s),\ (\varphi,\ P)) &= \begin{cases} m & , \text{ if } \ c \wedge \bigwedge_{p \in P} p \\ \alpha(s,\ (\varphi,\ P)) & , \text{ otherwise} \end{cases} \end{aligned}$$

where c is a Boolean combination of propositions in *Prop*, $m \in M$, $s \in Strat$ and $(\varphi,\ P) \in Formula_{RA}$.

In general *Prop* contains propositions stating if some properties hold or not. Let us collect some ideas, which properties could be relevant.

- Propositions about the Boolean structure of the formula. In our case it is always a conjunction of positive literals. If we generalize the definition of the formula, we could ask whether it is such a conjunction or not. We could also be interested in whether the formula is in conjunctive normal form or not. We are going to consider these properties in future releases of `SMT-RAT`.

- Propositions about the constraints in the formula. Does the formula (not) contain (only) equations/(strict inequalities)?
- Propositions about the polynomials compared by the constraint. Are they linear? Are they univariate? Do they contain more than n variables? Is their degree smaller than n?

There is another group of propositions, which are not that obvious but even essential for being able to use strategies to specify the backend of a module. They describe if a module is capable of solving the formula. Considering the definition of the strategy, we cannot guarantee termination without these propositions. A module, which we run on the formula which it cannot solve, would return `unknown` and be called on the same formula again and again, if we do not have these propositions. However, it is not easy to decide whether a module can solve a formula or not, before we just tried. Indeed, this is exact the way how we achieve it. We run a module on a formula and if it cannot be solved by the module, it marks the formula by a flag indicating this. By reason of efficiency we have also flags for all the other propositions. So, once the formula is generated all flags are set and we do not have to check, e.g. the degree of all the polynomials in it several times.

6.2.2 Specifying a strategy in the theory solver

The class `Manager` contains a strategy, which provides three methods to modify it.

- `bool addModuleType( const Condition, ModuleType )`

  Adds a module type for the backend being returned if the given condition holds and no previously added condition holds. It returns `true` if the strategy has been changed.

- `bool removeCase( const Condition )`

  Removes the case from the strategy, where the given condtion holds. It returns `true` if the strategy has been changed.

Note, that the strategy depends on the order of calling `addModuleType`. Thus, calling `addModuleType(`c_1`, `m_1`)` and then `addModuleType(`c_2`, `m_2`)` specifies the strategy

$$c_1 \ ? \ m_1 \ : \ (m_2)$$

and vice versa the strategy

$$c_2 \ ? \ m_2 \ : \ (m_1).$$

We provide the following self-explanatory propositions.

- PROP_TRUE
- PROP_IS_PURE_CONJUNCTION
- PROP_IS_IN_CNF

- PROP_IS_IN_NNF
- PROP_CONTAINS_EQUATION
- PROP_CONTAINS_INEQUALITY
- PROP_CONTAINS_STRICT_INEQUALITY
- PROP_CONTAINS_LINEAR_POLYNOMIAL
- PROP_CONTAINS_NONLINEAR_POLYNOMIAL
- PROP_CONTAINS_MULTIVARIATE_POLYNOMIAL
- PROP_VARIABLE_DEGREE_LESS_THAN_THREE
- PROP_VARIABLE_DEGREE_LESS_THAN_FOUR
- PROP_VARIABLE_DEGREE_LESS_THAN_FIVE
- PROP_CANNOT_BE_SOLVED_BY_SIMPLIFIER_MODULE
- PROP_CANNOT_BE_SOLVED_BY_GROEBNER_MODULE
- PROP_CANNOT_BE_SOLVED_BY_VS_MODULE
- PROP_CANNOT_BE_SOLVED_BY_UNIVARIATECAD_MODULE
- PROP_CANNOT_BE_SOLVED_BY_CAD_MODULE

If, e.g., the backend of type m has to be used, if none of the propositions specified in a strategy hold, you have to extend the strategy by an always fulfilled case returning m, i.e., call `addModuleType(PROP_TRUE, m)` as the last modification of this strategy.