A Short Documentation of Decision Procedures in KeY

Benjamin Niedermann

July 2, 2012

The purpose of this document is to give experienced KeY users and KeY developers a short introduction into the connection between decision procedures (e.g., Yices, Z3 and Simplify) and the KeY-system. As this connection mainly consists of the implementation of the SMT-format[1] we also call it the *SMT-integration* of KeY. The only solver that does not make use of that format, but uses a native format is Simplify, which is not maintained anymore by its developers. We therefore only consider the SMT-format in this document and also call the external solvers connected to KeY *the supported SMT-solvers*.

Further it is assumed that the reader is familiar with the KeY-system and its underlying dynamic logic. We therefore do not introduce common definitions, but we assume that the reader is aware of the most important definitions and notations as used in the KeY-Book. The document consists of the following parts:

1. **Basic Concepts** – It is explained how external solvers are connected to the KeY-System and which major design decisions have been made for the implementation.
2. **Using the SMT-Integration** – This section is addressed to readers that want to use the integration: It is explained how to use solvers within KeY and how to manipulate the settings of these solvers and the translation.
3. **Implementation** – In this part the structure of the SMT-integration of KeY is explained in more detail in order to give developers of the KeY-system a short introduction into the implementation of the SMT-integration.
4. **Case Study** – The case study describes roughly how counterexamples that are created by solvers can be used for deriving information about the currently considered sequent.

Note that this document is only written for purposes of documentation. It does not demand to be complete or to introduce new knowledge, but it should help to get in touch with the SMT-integration of the KeY-system.

1 Basic Concepts

In this section we shortly discuss how the SMT-integration basically works. Figure 1 illustrates the approach we are following: Assuming that we are given a sequent $S\colon \Gamma \vdash \Delta$, we create a first-order logic formula based on that sequent. To that end we introduce a function T that maps a formula Φ_{DL} of dynamic logic onto a formula Φ_{FOL} of first-order logic such that the following property is satisfied:

Property 1. *Φ_{DL} is valid if Φ_{FOL} is valid.*

[1] http://www.smtlib.org/

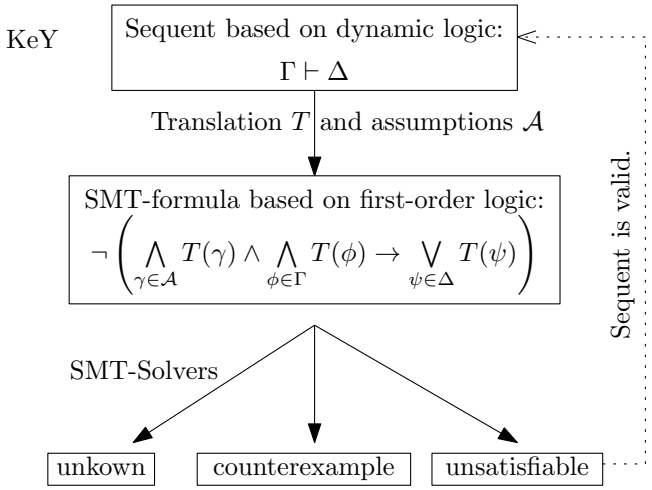


Figure 1: The structure of the SMT-integration.

On the implementation level the function T corresponds to a translation of Φ_{DL} into Φ_{FOL} based on the SMT-format.

As certain properties cannot directly be modeled by classic first-order logic (e.g., sorts) we introduce a finite set $\mathcal{A}$ of formulas that describe those properties. We call $\mathcal{A}$ the *additional assumptions* of T.

Then, we translate the sequent S into the following first-order logic formula

$$T(S) := \neg \left(\bigwedge_{\gamma \in \mathcal{A}} T(\gamma) \wedge \bigwedge_{\phi \in \Gamma} T(\phi) \rightarrow \bigvee_{\psi \in \Delta} T(\psi) \right) \tag{1}$$

whereas we call $T(S)$ *the translation of* S. This formula is then passed to the external solvers, which try to prove its unsatisfiability. There can occur three results:

1. **unkown:** The solver is not able to prove the unsatisfiability of $T(S)$, but it also cannot present a model proving the satisfiability. Roughly spoken, this event occurs either when the solver is overwhelmed or some timeout has exceeded.

2. **counterexample:** The solver has found a model satisfying $T(S)$. By Property 1 this does not yield necessarily that the given sequent is not valid. In Section 4 we give a short outlook how that counterexample can be used in order to obtain further information.

3. **unsatisfiable:** The formula $T(S)$ is unsatisfiable, which also implies that the given sequent is valid. The corresponding goal of that sequent can therefore be closed.

The translation T can be described recursively regarding the structure of a formula. For sub-formulas and terms based on first-order logic one can basically define T as the identity. On the implementation level this means a straightforward conversion from the format used by KeY into the SMT-format.

For sub-formulas and terms based on dynamic logic, we cannot use a direct translation, but we translate those formulas and terms into uninterpreted unique predicates and functions, respectively. To that end all free variables within the given term/formula are collected and used

as parameters for the arising predicate or function. For example the dynamic formula

$$[\pi]\, x = y$$

over the program π is translated into the first-order logic formula

$$\text{modOp1}(x, y)$$

where modOp1 is a new predicate symbol, which is only used for exactly that formula. Consequently, the translation T still satisfies Property 1.

As already mentioned, we also introduce a finite set $\mathcal{A}$ of assumptions that model concepts that are not supported by classic first-order logic. In the following paragraphs we describe which assumptions we have exactly implemented.

Sorts: In order to be independent from the concrete solvers, we do not make use of the type-systems that are offered by those solvers, but we model the sort-hierarchy explicitly by means of predicates. To that end we introduce for each sort X that occurs within the given sequent S an unary predicate *typeOfX* that indicates whether an object belongs to that sort or not. Then, in order to model the hierarchy we straightforwardly introduce implications that describe the relation between sorts: For example let A be the sub-sort of B and let B be the sub-sort of C, then we create the following two assumptions:

$$\forall x\colon \text{typeOfA}(x) \rightarrow \text{typeOfB}(x)$$
$$\forall x\colon \text{typeOfB}(x) \rightarrow \text{typeOfC}(x)$$

Finally, we need to enforce that for each type there is at least one object within the universe. To that end we create for each sort X within S the following assumption:

$$\exists x\colon \text{typeOfX}(x)$$

Unique Functions: As attributes of objects and arrays are modeled as unique functions within the KeY-system, we need to enforce the uniqueness of such functions within the SMT-integration. For that purpose we introduce for each pair

$$g\colon A_0 \rightarrow A_1 \times \ldots \times A_n, f\colon A_0 \rightarrow A_1 \times \ldots \times A_n$$

of unique functions of same signature the following assumption:

$$\forall x_1, \ldots, x_n\colon f(x_1, \ldots, x_n) \neq g(x_1, \ldots, x_n)$$

Taclets: Based on taclets optional assumptions can be introduced. To that end taclets are translated into valid formulas still encoding the information of that taclet. For a detailed description how the translation is implemented we refer to the documentation that can be found in *KEY_FOLDER/doc/smt/TacletTranslation.pdf*. In Section 2 it is then described in more detail how one can choose taclets.

2 Using the SMT-Integration

Table 1 shows the solvers which are supported by KeY. It is highly recommended to use the versions as stated, because only for those versions the integration is stable. For other versions it is known that errors can occur because the output of the solvers differs for those versions.

Solver	Version	Format
Z3	3.2	SMT2
Yices	1.0.34	SMT1
CVC3	2.2	SMT1
Simplify	1.5.4	Simplify-Format

Table 1: Supported solvers.

Figure 2 shows both the option dialog and the progress dialog of the SMT-integration. The former one can be reached by the menu entry *Options/SMT Solver*. By means of this dialog the user can customize the SMT-integration of KeY. The latter dialog pops up when solvers are started and shows the current progress. In the following we explain in more detail how to use these dialogs.

2.1 The Option Dialog

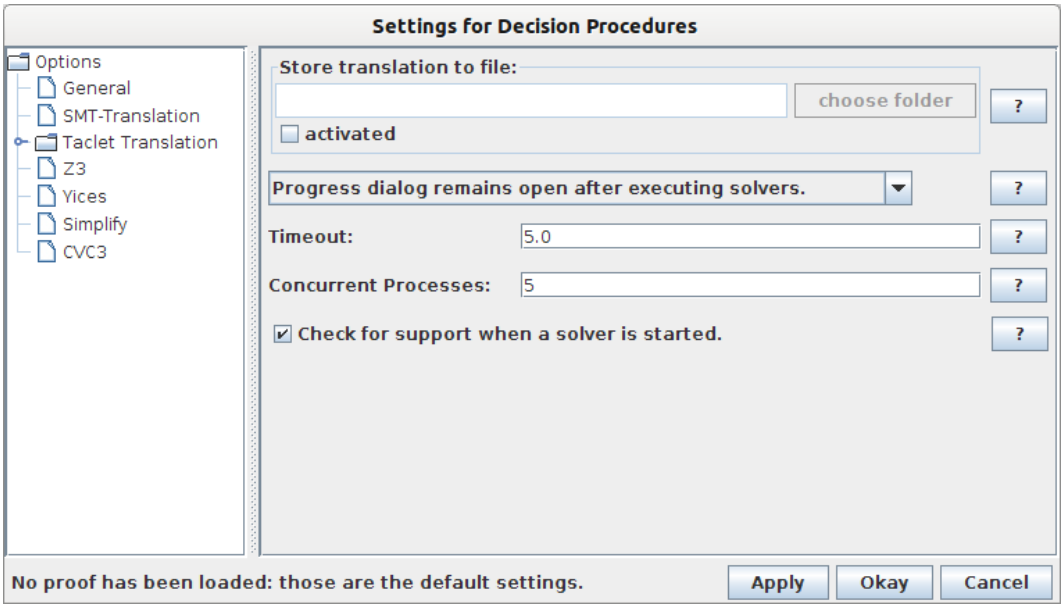
In this section we explain the single options which can be set for the solvers under *Options/SMT Solvers*. Mainly, there are two types of options. The first ones control the behavior of the solvers, while the second ones control the translation of a KeY formula into the corresponding format.

Controlling the Solvers: Since most of the options are already explained within the options dialog, we only explain the most important ones, which can be found in the tab *General*, in order to give further hints:

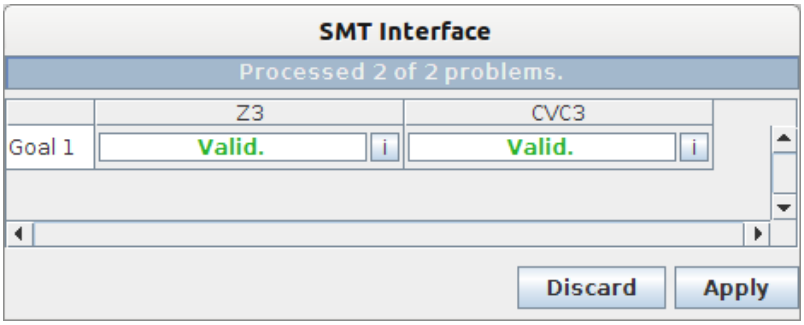
- **Timeout T**: Describes in seconds how long a solver instance may run until KeY interrupts its execution. Consequently, T does not state an overall timeout, but only a timeout for each run of a solver. That means that applying a solver on several goals, for each goal the same amount of time, namely T, is invested in order to solve the goal.
- **Concurrent processes**: While applying the SMT-integration of KeY, for each given goal and each solver exactly one external process is started (instance of a solver). This option specifies how many of those external processes may run concurrently.
- **Check for support when a solver is started**: If this option is set, each time a solver is started first it is checked which version of the solver is used. If the version does not coincide with one of the recommended versions a warning is presented in the progress dialog. Using a non recommended version can yield to an exception in the worst case, but it is still guaranteed that the soundness is not touched.

Further, for each single supported solver there is a tab showing the individual settings of those solvers:

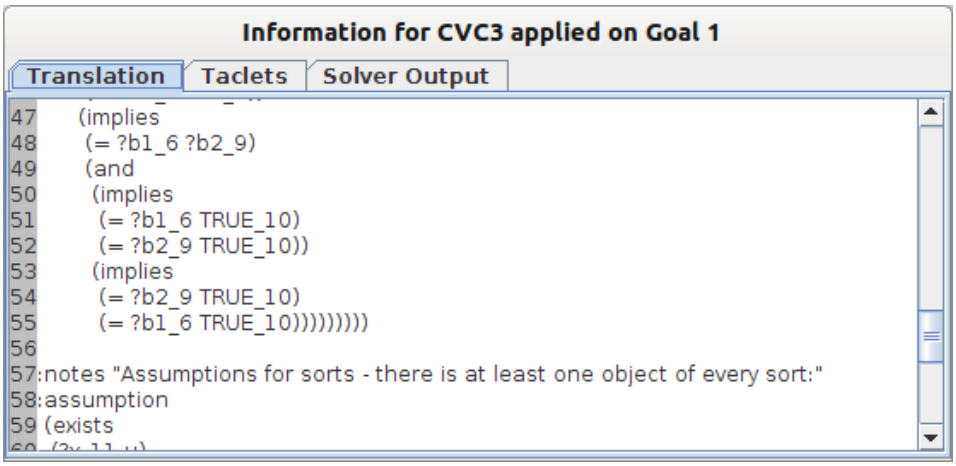
- **Installed**: This field shows whether KeY has recognized the solver. To that end KeY checks whether there exists a corresponding file within the folder specified by the command defined in the field *Command*. For example, for the command */home/user/z3/bin/z3* KeY checks whether there is a file z3 contained in the folder */home/user/z3/bin/*. If the command is changed, then by means of the button *Apply* the check mechanism can be started.
- **Command:** The starting command of a solver can be defined by means of an absolute path or by means of the path variable *PATH* of the operating system. In the latter case KeY searches in all folders defined by *PATH* in order to find the file defined in the command field.



(a) Option dialog.



(b) Progress dialog.



(c) Dialog for detailed information.

Figure 2: Dialogs related to the SMT-integration.

- **Parameters:** Specifies the parameters that are passed to the solver when it is started. As the default parameters are crucial for the stable execution of the solvers, it is recommended only to add parameters without removing the default parameters. However, the default parameters should usually be sufficient for executing the solvers.
- **Supported:** States whether the solver is supported by KeY, that is, it is of the correct version. To that end KeY starts the solver and checks for its version. By means of the button *Check for support.* the check can be triggered.

SMT-Translation:

- **Use explicit type hierarchy.** If this option is selected transitive relations in the type hierarchy are modeled explicitly. For example assume that A is a subtype of B and B is the subtype of the type C. In general it is sufficient to introduce the following two formulas as assumptions for modeling the type hierarchy:

$$\forall x\colon \text{typeOfA}(x) \rightarrow \text{typeOfB}(x)$$
$$\forall x\colon \text{typeOfB}(x) \rightarrow \text{typeOfC}(x)$$

  where typeOfX denotes the corresponding type predicate for a type X. However, the experience shows that some solvers are overwhelmed by the transitive relation $\forall x\colon \text{typeOfA}(x) \rightarrow \text{typeOfC}(x)$ such that they cannot derive it on their own. On that account the translation can be explicitly enriched by formulas also modeling transitive type relations. For the example this means, that the formula $\forall x\colon \text{typeOfA}(x) \rightarrow \text{typeOfC}(x)$ is also introduced as assumption if the option is selected.

  As this is done exhaustively, that is, for each transitive relation an assumption is introduced, for huge type hierarchies this leads to many extra assumptions. In more detail there can be $\mathcal{O}(n^2)$ assumptions when n denotes the number of types.

- **Instantiate hierarchy assumptions if possible.** This option can be used in order to simplify the type hierarchy and to resolve *obvious* relations. So far this option applies only for the null-object `null`: For all types $T_1, \ldots, T_n$ that extend the Java type `Object` the following assumptions are introduced:

$$\text{typeOf}\, T_1(\texttt{null}), \ldots, \text{typeOf}\, T_n(\texttt{null})$$

  Future Work: This approach can also be adapted to other constants used within the given sequent, such that for those constants the type hierarchy is *unrolled*.

- **Use built-in mechanism for uniqueness if possible.** As already mentioned, for translating attributes of objects and arrays correctly we need unique functions. To that end we introduce special assumptions as described in Section 1. However, some of the solvers (e.g., Z3) offer built-in mechanisms for unique functions, so that those can be used instead of additional assumptions. If this option is selected, exactly this approach is applied.
- **Use uninterpreted multiplication if necessary.** Some solvers support only certain kinds of multiplications. For example the solver Yices supports only multiplications of type $a \cdot b$ where a or b must be a concrete number. In order to support more complex multiplications, this option can be activated: If the solver does not support a certain kind

of multiplication, the occurrence of this multiplication is translated into the uninterpreted function *mult*. Moreover, the following assumptions are added:

$$\forall x\colon mult(x,0) = 0$$
$$\forall x\colon mult(0,x) = 0$$
$$\forall x\colon mult(x,1) = x$$
$$\forall x\colon mult(1,x) = x$$
$$\forall x \forall y\colon mult(x,y) = mult(y,x)$$
$$\forall x \forall y \forall z\colon mult(mult(x,y),z) = mult(x,mult(y,z))$$
$$\forall x \forall y\colon mult(x,y) = 0 -> (x = 0 \text{ and } y = 0)$$
$$\forall x \forall y\colon mult(x,y) = 1 -> (x = 1 \text{ or } y = 1)$$

Note:

1. If this option is not selected, an exception is thrown in the case that an unsupported multiplication occurs.
2. The translation makes the uninterpreted function *mult* unique so that there cannot be any name clashes with functions that are introduced by the user.

- **Use integers for too big or too small numbers.** Some of the solvers (e.g., Simplify) do only support numbers of fixed size. In the case that one uses less or greater numbers than the supported range allows, the solvers abort with an error. In order to circumvent this restriction this option can be selected:

  Numbers that are not supported by the used solver are translated into uninterpreted constants. Furthermore an assumption is added that states that the introduced constant is greater than the maximum of the supported numbers. Example: Assume that a solver supports numbers less or equal 10. Then the number 11 is translated into the constant c_{11} and the assumption $c_{11} > 10$ is introduced.

Taclet Translation: This tab allows the user to select certain taclets that are translated into valid first-order logic formulas that are passed as assumptions to the external solvers. To that end the taclets are divided into several categories. It is possible to select either all taclets or a sub-set of a category.

Further, some of the taclets also contain generic sorts. While translating the taclets, these generic sorts are instantiated with concrete sorts such that each possible instantiation is created. Consequently, if the given sequent contains many different sorts it can occur, that there are many different instantiations for the generic sort. In order to control the number, the user can specify how many generic sorts a taclet may have at most.

2.2 The Progress Dialog

The progress dialog pops up when a solver is started within the GUI-mode of KeY. It shows the current progress of the applied external solvers and presents the upcoming results (see Figure 2b). The dialog is structured by one progress bar showing the overall progress and a table containing for each goal and each solver exactly one cell: Each cell contains a progress bar and a button with title *i*. While the further one shows the current status of the given solver applied on that particular goal, the latter one becomes applicable when the solver has been executed on the goal: It triggers a dialog that shows detailed information (see Figure 2c):

- Translation: This tab contains the SMT-translation of the given sequent that is passed to the given solver.
- Taclets: If taclets are translated into assumptions, this tab contains the translation of these taclets.
- Solver Output: While applying a solver on a particular goal, the solver sends messages to the KeY-system. Those messages are presented within this tab.

The progress dialog also contains the two buttons *Discard* and *Apply*. The first one effects that the dialog is closed losing the results of the solvers, while the second one sets all goals to be closed for which the solvers could prove that the corresponding sequent is valid.

3 Implementation

In this section we describe the most important design decisions we have made. The SMT-Integration can be divided into two main parts: While one is responsible for the translation of a sequent into a SMT-formula, the other treats the execution of the solvers. We therefore have divided this section in two subsections considering these parts independently.

3.1 Translating a Sequent:

The structure of the classes used for translating a sequent $S\colon \Gamma \vdash \Delta$ is illustrated in Figure 3. The idea is that the translation is described on an abstract level by means of the class `AbstractSMTTranslation` such that the concrete translation is described by means of its sub-classes. To that end we assume that S is already given as a formula having the shape:

$$\bigwedge_{\phi\in\Gamma} \phi \rightarrow \bigvee_{\psi\in\Delta} \psi$$

In order to translate that formal into an representation that is understandable for the solvers, it is translated by `translateProblem` into a string representation using the specific format of the considered solver. The following listing sketches the most important method calls within `translateProblem`:

```
public StringBuffer translateProblem(Term problem,
                                     SMTSettings settings){

    StringBuffer smtProblem = translateTerm(problem,settings);

    List<StringBuffer> assumptions =
                        createAssumptions(problem,settings);

    builtCompleteText(smtProblem,assumptions,settings);
}
```

As the name suggest, the method `translateTerm` translates a term into the corresponding SMT-format. To that end it makes use of the *template method pattern*, that is, the abstract class `AbstractSMTTranslation` implements the method `translateTerm` such that some of the called methods within that method are overridden by the sub-classes of `AbstractSMTTranslation`. The following code snippet illustrates the structure of that procedure:

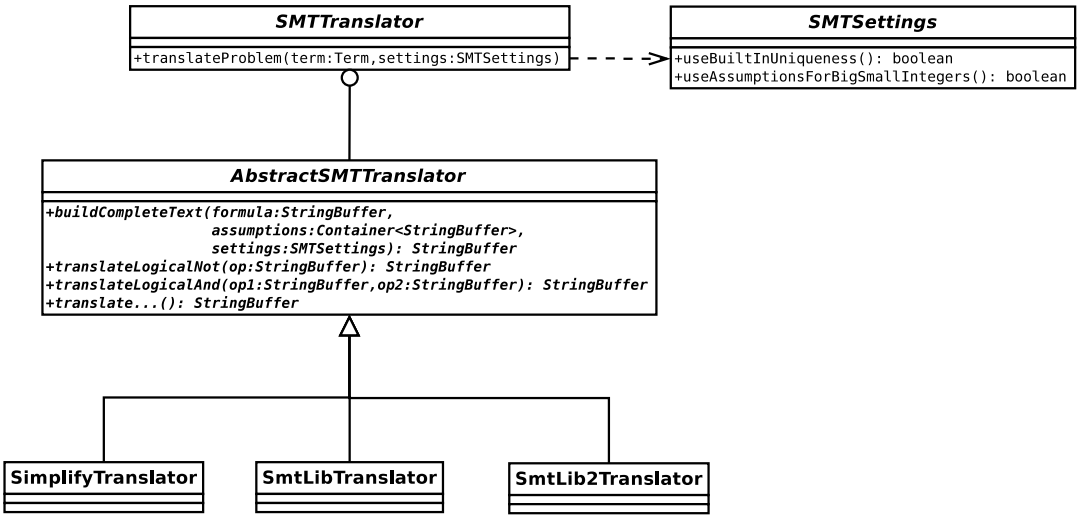


Figure 3: Class diagram of the most important classes involved in the translation. The list of methods of the classes is not complete

```
public StringBuffer translateTerm(Term term, SMTSettings settings){
   StringBuffer subTerms [] = new StringBuffer[term.arity()];

   for(int i=0; i < term.arity() i++){
         subTerm[i] = translateTerm(term.sub(i),settings);
   }

   if(term.op()== Junctor.AND){
     return translateLogicalAnd(subTerms[0],subTerms[1]);
   }else if(term.op() == Juncter.NOT){
     return translateLogicalNot(subTerms[0]);
   }else if(...){
     ...
   }
}
```

After translating the sub-terms recursively, the resulting translations of these sub-terms are assembled regarding the operator of `term`. For example, if the given term describes a logical and, then the abstract method `translateLogicalAnd` is called in order to assemble the sub-terms. How the logical and is translated in fact is not specified at this point, but it depends on the concrete implementation of the sub-classes of `AbstractSMTTranslation`.

The method call `createAssumptions` in `translateProblem` creates the assumptions as described in Section 1. To that end for each assumption an own string representation is created.

Finally, the method `buildCompleteText` is called, which builds the string representation of the problem that is passed to the external solvers. To that end it assembles the translations of the given assumptions and the translation of the given sequent. Further, it enriches the problem representation with settings that control the solvers (e.g., used logic, different strategies). As the creation of the complete problem representation depends on the considered format, this method is abstract and must be implemented by its sub-classes.

3.2 Executing External Solvers

Figure 4 shows the class diagram of the classes used for modeling the execution of external solvers within KeY.

In order to model a concrete solver with its properties we have introduced the interface `SolverType`: For each solver (Z3, Yices,...) there is a singleton implementing that interface. Those singletons mainly encode information about

- the versions that are supported by KeY, and
- the command that is used to start the solver, and
- the parameters that are additionally passed to the solvers.

Further, the method `createTranslator` allows to create instances of the interface `SMTTranslator` by means of the factory pattern, whereas an instance of the appropriate sub-class is returned (e.g., while `Z3Solver` returns an instance of `SmtLib2Translator`, `YicesSolver` returns an instance of `SmtLibTranslator`).

The method `createSolver` provides the possibility to create instances of the class `SMTSolver` by means of the factory pattern. An object of that class represents a single process of a solver. Consequently, there can be several instances of that class for the same solver at the same time. This class also contains an instance of the class `SMTResult`, which stores the result of that particular solver. Moreover, the constructor of `SMTSolver` expects an instance of the class `SMTProblem`.

The class `SMTProblem` encapsulates the problem to be considered, that is, it stores either a goal, sequent or formula. If it stores a formula then `getTerm` yields exactly that formula. Otherwise it creates a formula of shape $\bigwedge_{\phi\in\Gamma}\phi \rightarrow \bigvee_{\psi\in\Delta}\psi$ based on the given goal or sequent and returns it. Further, it contains a list that stores instances of the class `SMTSolver`. Later on, this list stores the concrete solver instances that have been applied on that problem.

By means of the class `SolverLauncher` the solvers can be triggered. To that end it contains the method `launch` that expects a list `problems` containing instances of `SMTProblem` and a further list `solvers` containing instances of `SolverType`. Calling `launch` leads to the execution of those solvers on all problems: To that end for each problem $P \in$ `problems` and each solver $S \in$ `solvers` one instance $I_{P,S}$ of `SMTSolver` is created and attached to P.

Afterwards, each instance $I_{P,S}$ is applied on the problem to which it is attached. To that end the method `start` of $I_{P,S}$ is called: In particular this lead to the translation of P into the proper format and the execution of an external solver that corresponds to S. In order to communicate with S while it is executed, a pipe is created between KeY and the solver (see class `Pipe`). By means of that pipe it is possible to send commands to the solver and to exert direct influence on the solver.

After the external process that is related to $I_{P,S}$ has stopped, the result is analyzed and stored within an instance of `SMTResult` that is attached to $I_{P,S}$.

4 Case Study

In the case that the considered sequent S is not valid, the translation $T(S)$ of that sequent is satisfiable, that is, there is a model that fulfills that formula. This model can then be used for analyzing why the sequent is not valid. Thus, we desire to obtain models for sequents that are not valid. Even though the solvers automatically generate those model if possible, it is not guaranteed that they can always find such a model. If the translation $T(S)$ is too complicated, the solver is overwhelmed and returns the result *unkown*.

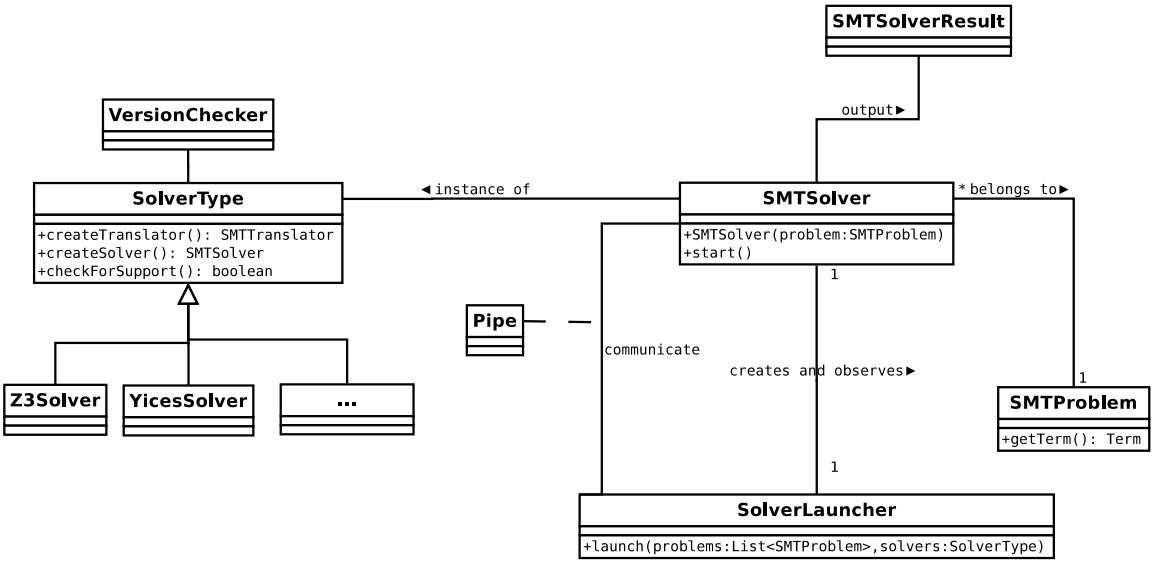


Figure 4: Class diagram of the most important classes involved in the execution of solvers.

On that account we are looking for an approach that simplifies $T(S)$ still preserving the property that we can draw conclusions about S. In this section we describe such an approach by presenting a case study.

The main idea is that we assume the universe to be finite. In particular we make the assumption that there are only few elements within the universe such that there is a manageable amount of instantiations of $T(S)$ that can be checked automatically. The hope is that even the restricted universe contains a model proving the satisfiability. Note that this approach only can be used for analyzing the sequent: It can occur that we find *spurious* models for $T(S)$, that is, the model satisfies $T(S)$ within the restricted universe, but not within the infinite universe. For example assume that the considered universe consists of integers. While in the infinite case the following formula is unsatisfiable, in the finite case it is satisfiable:

$$\exists x \forall y \colon y \leq x$$

The approach can therefore only be used in order to obtain hints why the sequent cannot be proved valid. Figure 5 illustrates how a sequent can be analyzed.

In order describe the universe systematically each variable $T(S)$ is replaced with a bitvector referring exactly to one element in the universe. That means, that the elements are numbered by the bitvectors. In order to restrict the size of those bitvectors we further partition the universe into different classes:

- $\mathcal{H} = \{$Contains all heaps.$\}$
- $\mathcal{F} = \{$Contains all fields.$\}$
- $\mathcal{O} = \{$Contains all objects extending the Java-class `Object`.$\}$
- $\mathcal{I} = \{$Contains all integers.$\}$
- $\mathcal{B} = \{$contains all booleans.$\}$

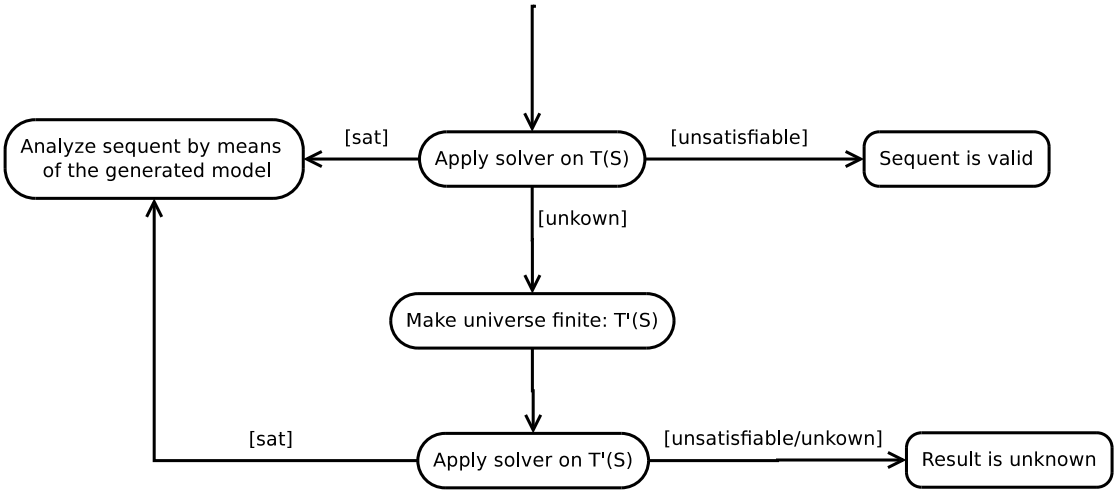


Figure 5: Analyzing a sequent by means of SMT-solvers.

For the different classes we can introduce different sizes for the used bitvectors. For example, while for $\mathcal{B}$ we only need a bitvector of size 1, we need for $\mathcal{I}$ larger bitvectors in order to gain appropriate results.

Now consider the following example[2]:

```
void sumAndMax(int[] a) {
  sum = 0;
  max = 0;
  int k = 0;

  /*@ loop_invariant
    @   0 <= k && k <= a.length
    @   && (\forall int i; 0 <= i && i < k; a[i] <= max)
    @   && (k == 0 ==> max == 0)
    @   && (k > 0 ==> (\exists int i; 0 <= i && i < k; max == a[i]))
    @   && sum == (\bsum int i; 0; k; a[i])
    @   && sum <= k * max;
    @
    @  assignable sum, max;
    @  decreases a.length - k;
    @*/
  while(k <= a.length) {
      if(max < a[k]) {
          max = a[k];
      }
      sum += a[k];
      k++;
```

[2]The example is based on a problem presented at the VSTTE'10 competition.

```
        }
    }
```

Due to condition `k <= a.length` the given program is not correct, but it leads to an index-out-of-bounds exception. On that account KeY cannot prove that program to be sound, but some branches remain open. For the following we assume that the file

KEY_FOLDER/examples/smt/casestudy/SumAndMaxProof.key

is loaded within KeY: Consider the open goal with number 2395: It occurs after splitting the proof for checking for an index-out-of-bounds exception such that all updates have already been resolved. When we apply Z3 on that goal, the solver stops because it exceeds the timeout. In the following we explain how we can use the translation that is passed to the solver in order to find a model.

In the file

KEY_FOLDER/examples/smt/casestudy/translation/original.smt

the original translation of that goal is stored (In Section 2 it is explained in more detail how to excerpt that translation from KeY). In order to introduce a finite universe we replace the two sorts u and Int with placeholders $\#BIT\_HEAP$, $\#BIT\_FIELD$, $\#BIT\_OBJECT$ and $\#BIT\_INT$ that represent bitvectors. We have stored the result in file

KEY_FOLDER/examples/smt/casestudy/translation/template.smt

For example in line 81 we have replaced the following function declaration

```
(declare-fun boolean_col__col_select_21
(u u u ) u )
```

with

```
(declare-fun boolean_col__col_select_21
(#BIT_HEAP #BIT_OBJECT #BIT_FIELD ) #BIT_OBJECT )
```

because by definition of the function *select* it expects a heap, an object and a field, and returns an object.

Further, in line 30 we have declared that the function *type_of_any_23_24* expects a bitvector modeling objects. As this bitvector may have a different size to bitvectors modeling integers, heaps and fields we need to comment out the assumptions in line 223, 231 and 260. These assumptions describe that integers, fields and heaps are sub-sorts of *any*, which we do not need to model necessarily.

By means of the program replace.jar, which is contained in

KEY_FOLDER/examples/smt/casestudy/translation/

we replace these placeholders with concrete values. To that end we execute the program within that particular folder by means of the following command:

java -jar ./replace.jar ./template.smt ./instantiation.smt

The program expects the user to enter the sizes of the different bitvector types. Appropriate values are:

$$BIT\_INT = 3$$
$$BIT\_HEAP = 2$$
$$BIT\_FIELD = 4$$
$$BIT\_OBJECT = 5$$

Then the program reads the file *./template.smt* and replaces all occurrences of the placeholders with the corresponding values. Further, it translates all numbers into bitvectors of corresponding size and replaces the operators <, <=, >=, > with *bvslt*, *bvsle*, *bvsgt* and *bvsge*, respectively. Finally, it stores the result in *./instantiation.smt*.

Afterwards, we start Z3 entering the following command in the terminal:

Z3 -in -smt2

The parameters say that we want to start Z3 using the interactive mode such that it expects the SMT2-format. We then copy the content of *instantiation.smt* into the terminal and confirm. After a short period of time the solver returns *sat*, which means that it has found a model.

The command

(get-value k_0_8)

returns the value for the variable `k` and the command

(get-value ((length_7 a_2)))

returns the length of the array `a`. In both cases the same bitvector is returned, that is, we have detected an index-out-of-bounds error.