# Supporting Loop Proofs in KeY by using BLAST

Mathias Krebs


Technical Report No. LGL-REPORT-2006-004
May 2006

Software Engineering Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland

# Supporting Loop Proofs in KeY by using BLAST[1]

Mathias Krebs

**Abstract**

In contrast to beta-testing, formal verification can guarantee correctness of a program against a specification. Two basic verification techniques are theorem proving and model checking. Both have strengths and weaknesses. Theorem proving is powerful, but difficult to use for a software engineer. Model checking is fully automatic, but less powerful and hard to extend. This paper shows a possibility, how to combine both approaches, in order to surround the weaknesses. Concretely, we automatize the proof of while loops in the theorem prover KeY, by using the model checker BLAST.

[1] This is a revised version of M.Krebs' diploma thesis, written at EPFL and finished in February 2006.

# Contents

# 1 Introduction

## 1.1 Motivation

Driven by the high number of bugs in industrial and consumer software, computer scientists try to find better methods in quality assurance. The mainstream approach today is testing. Elaborated beta-testing procedures are known, but there is a principal problem. As Djikstra said, testing can never show the absence of errors, only that there are errors. It's not possible to tell if software is correct by testing, because the fact not to have found errors does not guarantee an error free program.

Software engineering is more than writing program code nowadays. Doing several refinement steps before writing the code is standard. Interesting for our purpose is that developers often specify the code. A usual specification expresses what we expect a function to do under which conditions. It could be a text written in a natural language such as English. Unfortunately, this leaves room to ambiguities and losses. A more scientific approach consists in using a formal language such as OCL [11]. Another possibility is to use higher order logic languages, which are well known to mathematicians and computer scientists.

If we have program code and a formal specification of what it should do under which conditions, we have set the base for formal verification. Notice that when doing verification, we can not exclude all errors. If there is an error in the specification, we cannot discover it using logic.

Two groups of tools can be identified today in the formal method community. One group uses the theorem proving approach, the other the model checking approach. In one sentence we can say, that theorem proving is more powerful, but harder to use than model checking.

A sophisticated theorem prover is the KeY System [10]. It provides a comfortable user interface for proving. KeY focuses on the JAVA language, more precisely on a subset called JAVA CARD [6]. Additionally, a version for C is in development, but we focus on the JAVA version.

A major source of difficulties for KeY users are loop constructs. If the number of iterations is not predetermined in the code, all available heuristics fail. The reason of failure is, that the program cannot be entirely unrolled, given the number of iterations is unknown. Two ways to overcome this problem exist, the induction method and the invariant method. We use the invariant method in this paper. However, we believe that knowledge on invariants can also be used for induction.

The basic idea of our approach is to use another tool, BLAST, to discover the invariants of a loop. This tool is often able to find automatically the solution of a problem, even when the heuristics of KeY fail. We can benefit of that by finding the invariants, and apply the knowledge in KeY. The advantage of our approach in contrast to directly using BLAST is the fact, that we can show total correctness, and not just partial correctness (see section 2.4, to learn more about correctness).

## 1.2 Related Work

In summary, we have created a new formal theory for an expressive temporal logic and used it to develop concrete technology to demonstrate that using a theorem prover as a tool programming platform provides us with several theoretical advantages without too high a performance penalty. We thus hope that this work will be of interest to the research community and also be of use to industrial practitioners. The approach developed in [1] combines model checking and theorem proving for an expressive temporal logic. The project focus is the integration of model checkers and theorem provers in general, rather than the development of techniques exploiting such an integration. A formal theory of the modal $\mu$-calculus was developed as theoretical support. The implementation was done for the *HOL* theorem prover.

Another combination is outlined in [2]. The tool *prioni* combines a model checker and a theorem prover in the following way. First, the specification is tested by model checking, eventual specification errors can be eliminated. Afterwards, a proof attempt on the refined specification using the theorem prover can be started. If the attempt is not successful, but a part of the problem is solved, it is possible to continue with model checking for the rest. The main benefit of *prioni* is, that it helps the user checking his specification, before he does the proof. On the other hand, no support is provided for theorem proving by the model checker.

## 1.3 Outline

Our paper is divided into four major parts. In chapter 2, we explain BLAST and KeY. Additionally we provide some fundamental knowledge on formal verification. This section does not cover topics specific to this paper, but gives a general introduction. The reader can skip the section or parts of it, if he knows the basics of KeY and BLAST. In chapter 3, we explain how we combine the model checking with the theorem proving paradigm. We introduce our idea and give the theoretical argumentation justifying our approach. No practical questions on implementation are mentioned here. For that purpose, the reader is referred to chapter 4. Here, we give an overview of the architecture of our plugin, and discuss implementation features, related to the KeY framework. In the appendix in chapter 7, we summarize the most interesting program examples we created, to develop and control our ideas.

# 2 Background of Tools and Theories

## 2.1 Basics of Formal Verification

Formal verification can be applied, if a specification and an implementation of a program is given. It is then possible to check, if the implementation obeys or violates the specification. We specify program code by defining a precondition and a postcondition for each function [2]. This kind of specification is known under the name *contracts* in the literature [12]. The precondition defines the valid initial states, the postcondition defines a warranty on the expected outcome. The example of a division function (see figure 1) illustrates this concept.

**precondition:** $b \neq 0$

```
double divide (double a, double b)
{
   int c = a/b;
   return c;
}
```

**postcondition:** $c \cdot b = a$

Figure 1: Division function and specification.

A division is only defined, if the divisor is not zero. More precisely, the behavior of the implementation is specified only for the case the divisor is different from zero. The division result can be checked by multiplying the result with the divisor (notice, that in practice, round-off errors can occur).

If we have a specification of this kind, there are no ambiguities left. Formal techniques can be applied, to determine if the program satisfies the postcondition, given the precondition is respected. An overview on formal methods is given in [8].

## 2.2 Theorem Proving

### 2.2.1 Introduction

Theorem provers use the same approach as mathematicians, when they prove something. They rely on a set of rules, which are given, and apply them in a clever way. The set of rules one needs to prove correctness of software extends those used by mathematicians, because the knowledge about the functionality of programs has to be encoded.

Another point is that formal proofs found by a theorem prover are much more detailed than a corresponding proof given by a mathematician. Experiments

[2] A function is also called procedure, routine or method in programming language theory. In our context, we use the term *function* in this sense.

carried out with the ILF system have shown, that on proof step done by a mathematician corresponds to ten steps in a formal proof.

Let's introduce here a sample problem, that will help to understand the ideas and the differences between the mathematical and the formal approach of theorem proving. We want to prove the following statement for natural numbers.

> If we assume $x = 0$ or $y = 0$, we can conclude that $x \cdot y = 0$, if $x$ and $y$ are natural numbers.

There are two cases to distinguish, $x = 0$ and $y = 0$, because we assume that only one of them has to be true.

1. If we assume $y = 0$, we can replace $y$ by zero in $x \cdot y = 0$, and we obtain $x \cdot 0 = 0$. By definition of the multiplication of natural numbers, we know that $a \cdot 0 = 0$ is true for every natural number $a$.

2. If we assume $x = 0$, we can replace $x$ by zero in $x \cdot y = 0$, and we obtain $0 \cdot y = 0$. Because we know, that $a \cdot b = b \cdot a$ in the context of natural numbers, we are allowed to rewrite the problem as $y \cdot 0 = 0$. In the same way as we do in case 1, we conclude this is true.

Most mathematicians would accept such a proof, because we use a precise language and note things properly. However, we don't use any convention that would help to guarantee correctness. We rely on the fact, that a person can understand and verify the proof.

### 2.2.2 Formal Proving

As mentioned before, mathematicians do not prove theorems formally. This does not mean, that their work is incorrect. They just omit steps, because it's easier this way to concentrate on the problems of their domain.

However, we want to execute proofs mechanically. In consequence, we don't omit intermediate steps. In order to show the difference, we prove the sample from section 2.2.1 again, this time in a mechanical way. First of all, we write the proof goal in a precise way, using first order logic.

$$\forall x \in \mathbb{N}\, \forall y \in \mathbb{N}(x = 0 \vee y = 0 \Rightarrow x \cdot y = 0)$$

Let's decode this expression. Every notation we use is explained below.

| | |
|---|---|
| $\forall x \in \mathbb{N}\, \forall y \in \mathbb{N}(...)$ | The phrase $\forall x \in \mathbb{N}$ means, that x is an arbitrary natural number. We can read it as " *for all x, x being an natural number*". This operator belongs to the group of the quantifiers. We quantify $y$ in the same way, because we want to say that both are arbitrary natural numbers. |

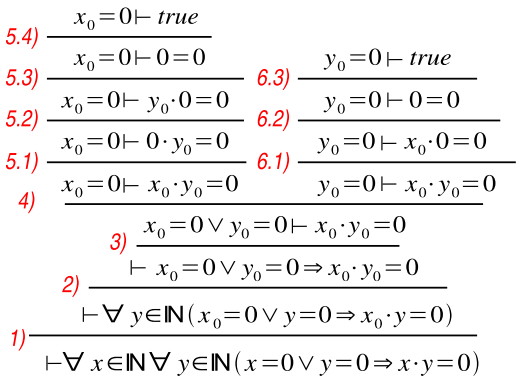


Figure 2: Formal proof of $\forall x \in \mathbb{N}\, \forall y \in \mathbb{N}(x = 0 \vee y = 0 \Rightarrow x \cdot y = 0)$.

| | |
|---|---|
| $\ldots \Rightarrow \ldots$ | We have to decompose the expression $x = 0 \vee y = 0 \Rightarrow x \cdot y = 0$ in which $\Rightarrow$ is the operator with the highest priority. On the left hand side of the arrow is premise, on the right hand side the conclusio. In natural language, we might say "*if* $x = 0 \vee y = 0$ *is true, we can conclude that* $x \cdot y = 0$ *is also true*". |
| $\ldots \vee \ldots$ | $x = 0 \vee y = 0$ means that at least one of the two, $x = 0$ and $y = 0$ is true. It corresponds to term "*or*" in the English language. |

For proving a goal, we will use in this thesis the *sequent calculus*. The same calculus is also used by the KeY system. The proof goal and any other intermediate results of proof steps have always the form of a sequent.

$\Gamma$, **list of hypothesises** $\vdash$ **list of goals**

The symbol $\vdash$ is also known as the *sequent symbol*, the list of hypotheses as antecedent, the list of goals as succedent.

For the investigated example, we can write the proof goal below.

$$\Gamma \vdash \forall x \in \mathbb{N}\, \forall y \in \mathbb{N}(x = 0 \vee y = 0 \Rightarrow x \cdot y = 0)$$

In our example, the list of hypothesis is empty. As we will see below, all axioms on natural numbers [3] that are necessary to prove the goal are encoded by the proof rules we use.

We can transform the proof goal by applying proof rules. In general, such a transformation should make the remaining goal simpler. Certain transformations can split the proof by generating more than one sub-goal. Therefore, the

[3] An example of such an axiom is the fact, that multiplication of any number with zero results to zero.

final has the form of a tree, a proof goal on every leaf. The initial proof goal is proven, if every leaf of the corresponding tree is equal to *true*. The art of formal verification consists in applying the proof rules leading to success. Figure 2 shows a complete proof tree for the example of this section.

Every rule can be expressed, using the following, formal notation. If we can match the current goal with the expression below a separator line, it is possible to transform it into the expression above the line.

In order to keep the rules flexible we use the two wildcard symbols. $\Gamma$ denotes an arbitrary list of hypothesises, $\Delta$ an arbitrary list of goals.

In the following, we explain the meaning of every rule application on figure 2. Notice that there is a huge set of such rules, that we don't introduce here. A systematic introduction into the field can be found in [13]. We give an informal and the formal description of each rule we apply.

| | | |
|---|---|---|
| 1, 2) | **all_right** We can transform $\forall x \in \mathbb{N} A$, by substituting x is with new term. We denote this term by sk, because it is often called Skolem term. [4] | $\frac{\Gamma \vdash A[x/sk], \Delta}{\Gamma \vdash \forall x \in \mathbb{N}\ A, \Delta}$ |
| 3) | **imp_right** We get rid of the arrow, by moving the expression on its left-hand side (premise) to the hypothesis list. The expression on the right-hand side (conclusio) remains within the conclusions to prove. This step moves the assumptions about $x$ and $y$ explicitly into the list of hypothesis. | $\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}$ |
| 4) | **or_right** Because we have an expression containing an or-operator in the hypotheses, we must split the proof. Both cases, $x = 0$ and $y = 0$, have to be treated separately. | $\frac{\Gamma, A \vdash \triangle \quad \Gamma, B \vdash \triangle}{\Gamma, A \vee B \vdash \triangle}$ |
| 5.1, 6.1) | **apply_equality** $x \cdot y$ can be replaced by $0 \cdot y$, because $x = 0$ is a hypothesis. | $\frac{\Gamma, a{=}b \vdash B[a/b], \Delta}{\Gamma, a{=}b \vdash B, \Delta}$ |
| 5.2) | **mul_comm** $0 \cdot y$ can be exchanged with $y \cdot 0$. We do this, because the following taclet is defined this way. | $\frac{\Gamma \vdash A(b \cdot a), \triangle}{\Gamma \vdash A(a \cdot b), \triangle}$ |
| 5.3, 6.2) | **times_zero** $y \cdot 0$ is equal to zero by definition of the multiplication. $y \cdot 0 = 0$ can be replaced by $0 = 0$. | $\frac{\Gamma \vdash B[a \cdot 0/0], \Delta}{\Gamma \vdash B, \Delta}$ |
| 5.4, 6.3) | **equal_literals** A claim of the form $a = a$ is always true. | $\frac{\Gamma \vdash true, \Delta}{\Gamma \vdash a{=}a, \Delta}$ |

[4] We use the notation $B[a/A]$ to express, that all free occurrences of the variable $a$ in $B$ are replaced by $A$. An occurrences is free, if the variable is not quantified.

The formal procedure causes more work, but has the advantage that it can be done mechanically, sometimes even automatically by a computer. We don't use hidden assumptions, at the cost of a detailed notation.

### 2.2.3 Dynamic Logic

So far, we did not touch the field of software verification by theorem proving. The key to this technique is dynamic logic[3]. It allows to use the power of the classical approach for proving program code. For that purpose, we introduce a new symbol, the so called diamond "$< \{ \} >$". Enclosed by that gem, we can write the program code to prove, and behind it the postcondition to satisfy. We rewrite the division function of section 2.1 in the following way, using dynamic logic.

$$\begin{array}{l} \vdash \forall a_L : a_L \in \mathbb{N} \Rightarrow \forall b_L : b_L \in \mathbb{N}, b_L \neq 0 \\ \Rightarrow \\ \{a := a_L\}\{b := b_L\} < \{c = a/b;\} > \\ (c \cdot b_L = a_L \vee c \cdot b_L = a_L - 1) \end{array}$$

Because program variables are integers by definition, we allow in the postcondition the case of a round-off error explicitly. A very important point is the distinction between logical and program variables. Using logical variables (denoted by an L-index here), we can define the precondition. By the mean of the so called updates, the program variables are initialized using the logical variables. Updates are enclosed by brackets. It is possible, to apply proof rules on a program enclosed by the diamond. Changes on program variables are tracked by the updates, which represent the current state of a variable. When the program has been rolled out completely, we assign the actual values of the program variables to the variables in the postcondition. The remaining proof goal is a first order logic expression without a diamond operator. Given the program satisfies the contracts, it is possible to show correctness using Logic as introduced in section 2.2.2.

### 2.2.4 KeY

KeY is a theorem prover suite, supporting a subset of the JAVA language. The exact specification of the subset is given in [6]. Three groups at the universities of Karlsruhe, Koblenz-Landau and Chalmers are developing the system. It provides an integration in Borland's Together CASE tool. KeY has a graphical user interface, that helps the user to execute a proof (see figure 3).

Proof rules are encoded as taclets in KeY. New taclets can be introduced by developers, as well as by the user. The asset of the taclet system is its flexibility. Basically, there are two possibilities on how to apply proof rules. The user can apply taclets by hand, using the interface. An automatic mode is available, too. The mode does not provide entire proof strategies, but it relies on heuristics.

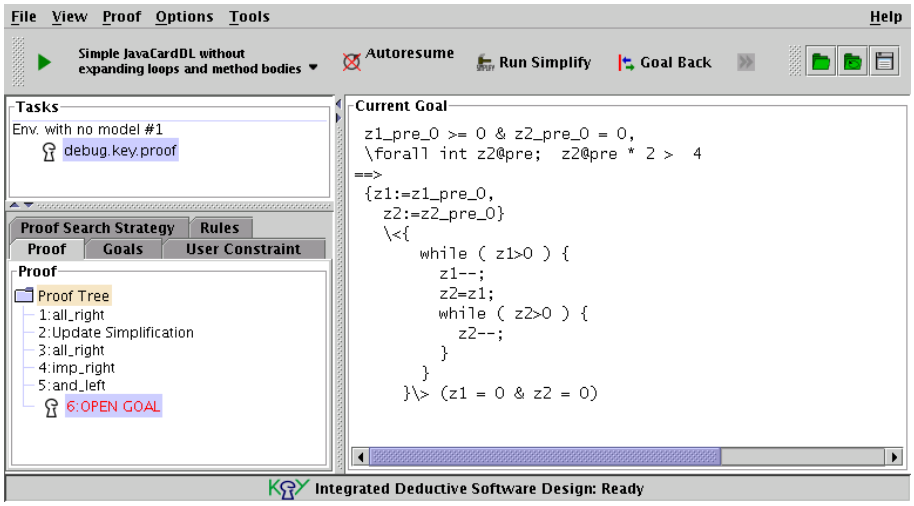


Figure 3: Interface of the KeY System.

The heuristics are powerful, but they show weaknesses, if quantifiers are in use. This is not amazing, because theoretical problems exist related to automatic solution-finding of quantified proof goals. Another problem for the heuristics are loop constructs in programs. The automatic mode is only successful, if the number of loop-iterations is predetermined in the program. However, in case the heuristics fail, the user can still try to solve the problem using the interface.

The theorem prover *Simplify* is integrated in KeY. *Simplify* is specialized on arithmetic problems and first-order logic. It's fully automatic and may help the user to close a goal, even if the heuristics fail.

The interface of KeY is composed of multiple panes. It shows the actual goal to prove, as well as an overview of the whole tree. Very often, proofs and the corresponding trees get complicated. Therefore, the interface provides the possibility to expand and hide subtrees.

In figure 4, we compare the proof tree of KeY with the classic tree we elaborated at section 2.2.2. For detailed information on KeY, we recommend [10].

### 2.2.5 Taclets in KeY

The goal of this section is to give an overview to the reader on the concept and the usage of taclets. To illustrate this, we revisit the example we already proved twice in this chapter. We don't give the entire solution, but we present the most interesting steps. First, we have to create a KeY problem file. Here is its content.

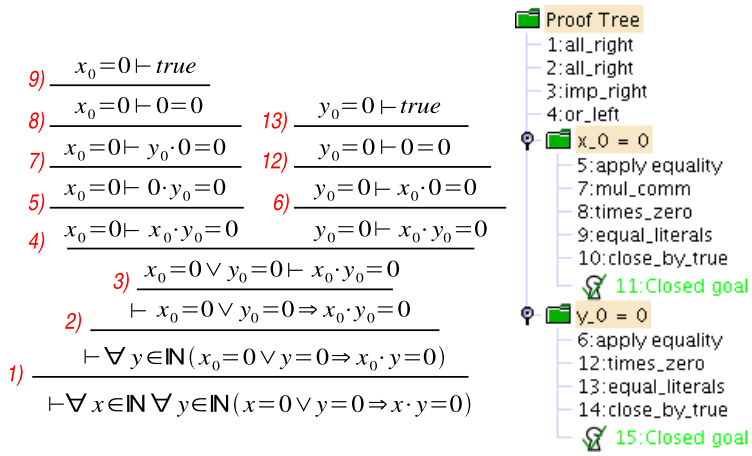


Figure 4: Classic proof tree vs. overview tree in KeY. The numbers correspond.

```
\problem
{
   \forall int x;
   \forall int y;
   (
      x=0 | y=0 -> x*y = 0
   )
}
```

**The *all_right* Taclet** The aim of the taclet is to eliminate the ∀-quantifier. The basic idea consists in replacing an all-quantified variable $x$ by an $x_0$, representing a new constant symbol having the same domain as $x$. The taclet is encoded in the following way in KeY taclet syntax.

```
all_right
{
   \find ( ==> \forall u; b )
   \replacewith { ==> {\subst u; sk}b }
}
```

The ==> symbol is equivalent to ⊢, introduced in section 2.2.2. The *find*-keyword specifies the situation in which the taclet can be applied. *all_right* looks for an expression $b$ in the succedent, quantified by a variable $u$. If the matching engine finds such an expression, it can be replaced by the expression $b[u/sk]$. In other words, the subst-keyword indicates a possible replacement of u by sk.

If user input is demanded in a taclet, the interface provides a pop up window to specify the input. In figure 5, we show the impact on the goal and the situation

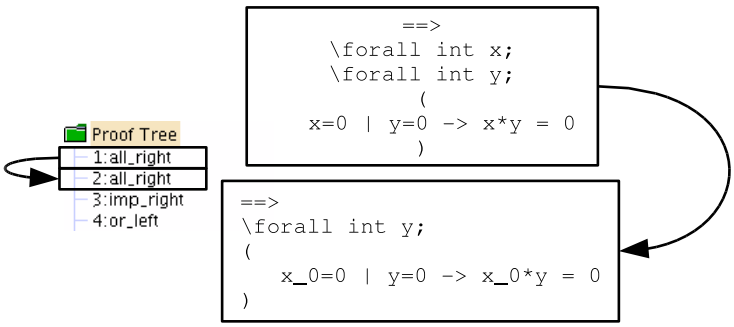


Figure 5: Application of the *all_right* taclet.

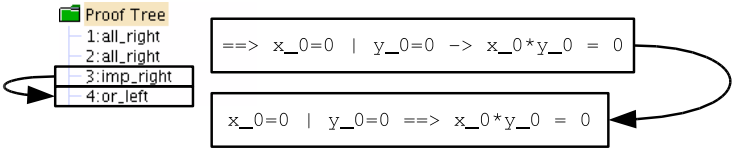


Figure 6: Application of the *imp_right* taclet.

in the overview tree. Notice, that we instantiate $x$ by $x\_0$ in our example.

**The *imp_right* Taclet** The aim of this taclet is to move the premise of an implication explicitly to the list of hypotheses. Its representation in taclet syntax is

```
imp_right
{
   \find ( ==> b -> c )
   \replacewith {b ==> c }
}
```

The lookup pattern is $==> b- > c$, the replacement option $b ==> c$. The impact of the rule application on the goal and on the tree is given in figure 6.

**The *or_left* Taclet** The taclet can be applied, if a term in the hypotheses contains an or-operator as top-level operator [5]. In such a situation, the proof can be split into two sub-goals, one for each sub-term of the term containing the or-operator. The corresponding taclet is encoded in the following way.

[5] A top-level operator is the operator, that has to be evaluated first in a term, given the operator priorities. In usual arithmetics for example, the top-level operator of the expression $a + b \cdot c$ is the addition-symbol.

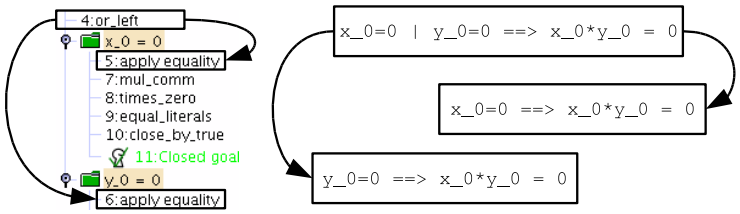


Figure 7: Application of the *or_left* taclet.

```
or_left
{
   \find ( b | c ==> )
   #c { \replacewith {c ==> } };
   #b { \replacewith {b ==> } }
}
```

The new property of this taclet is, that a rule application may create two or more subgoals. The #-symbol allows to specify a name for the sub-goal. Here the name is simply the content of b and c. Figure 7 shows the impact of the taclet application.

## 2.3 Model Checking

### 2.3.1 Introduction

For each program, we can define a state space. A state describes the status of the program execution at one moment. The status can be expressed by the actual location in the code, and by a set of assertions on the program variables. Such an assertion is an abstraction of the concrete system status. All possible states together form the state space. The execution of a program can be seen as a trace in the state space. If control structures such as loops are in the program, there may be an infinite number of traces. However, techniques based on the state assertions allow to keep the state graph finite.

Model checkers assemble all possible traces of the program in a graph. Given such a graph, it's possible to check if a trace leads to a dangerous state, or if the program is safe in the sense, that the error state is not reachable (see figure 8).

Let's consider as an illustration a dangerous state in a UNIX system.

- The current code location is the beginning of the routine, granting root access to the user.
- The assertions on the program variables indicate, that the root password has not been specified.

dangerous state unreachable  dangerous state reachable

Figure 8: Two state space diagrams. The dangerous state is represented by the skull.

If the reachability analysis does not find such a state on one of the possible traces, we can guarantee the program is safe in the sense, that the root password is always specified when root access is granted.

### 2.3.2 The BLAST Project

BLAST (Berkeley Lazy Abstraction Software Verification Tool) is a model checker engine written in Ocaml. The target programming language of the system is C. The simplest possible use of BLAST is the check of reachability of a specific error label in the source code. Additionally, a specification language exists. The goal of that language is to allow separation of specification from the source code. The interested reader is referred to [5] for detailed information.

### 2.3.3 How BLAST works

**Control Flow Automaton (CFA)** BLAST doesn't work directly on the code, it transforms the source first into the CFA. A CFA is an automaton, representing the control flow of the program. The control flow shows in which order the program locations are executed. A program location is basically a line number, but it's important to notice the instruction on that line has not been executed so far. It refers to that moment in the execution, just before the corresponding statement is executed. The CFA relates the program locations by arrows. An outgoing arrow means that from the current state we can go to the state the arrow points to. Arrows are annotated by the executed action at the corresponding code line. If the program location represents a condition test, for example an if-construct, the control flow is split. Arrows are annotated by predicates, indicating if the condition evaluates to true or to false. If the condition is not atomic[6], we have more than just one state in the automaton for that condition evaluation, because BLAST treats the atomic conditions individually. On the other hand, BLAST resumes a sequence of basic instructions[7] by one arrow. We give an example of a CFA construction in figure 9.

[6] In our context, atomic means that the expression does not contain the or-operator and the and-operator.

[7] A basic instruction is always an assignment in our context, for example $i = i - 1$.

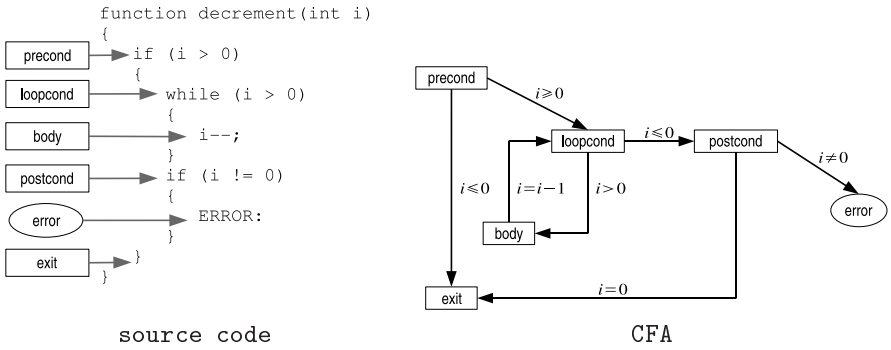


Figure 9: Source code and CFA of a simple decrement C program.

**Abstract Reachability Tree (ART)** The basic concept of the reachability analysis is the ART. Contrary to the CFA, the ART is a tree and not an automaton. It contains all possible execution traces of the CFA.

Every node of the ART can be annotated by an assertion on the program variables, representing the environment at this moment.

A leaf of a complete ART satisfies one of the following conditions.

1. It corresponds to a final state in the CFA. (such as exit in figure 9)
2. Its assertions on the program variables are contradictory.
3. It has the same status like an internal node (same CFA state, same or weaker assertions).

The second condition reflects the fact, that we never go in a contradictory state if we execute a program on a computer.

ARTs may remain finite. If a node represents the same program location as an internal node, and has the same or a weaker environment, the remaining trace is the same as for the internal node and can be ommited. In this paper, we use dashed lines on our figures to represent this.

Because of the third condition, an ART may remain finite, even if an infinite number of traces exist. If we unroll the CFA without that rule and the program contains a loop, the length of some traces would grow towards the infinite. However, the rule exists. BLAST notices, if the leaf's status is equal to the status of an internal state. In such a situation, the continuation of the trace may already be covered by an internal state.

The final ART is constructed iteratively. The CFA is unrolled until an error state is reachable, or until the ART is complete. If an error state is reachable in the ART, an abstraction refinement based on Craig Interpolation is launched. Two outcomes are possible. The error may represent a real error, or it has

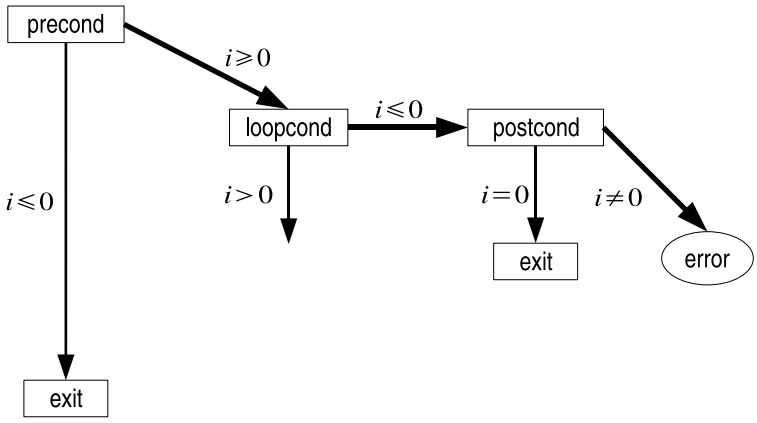


Figure 10: First ART of the simple decrement problem.

occurred because of an insufficient abstraction. In the later case, the Craig interpolation allows to enhance the abstraction by delivering better environment assertions.

**Craig Interpolation** If the error state is reachable in the ART, a refinement procedure tests by *counterexample guided abstraction refinement*, if the trace is feasible. It is possible, that a real error has been found, or the trace may exist, because the abstraction [8] is not good enough. The refinement states that the error is real, or it finds a better abstraction that excludes the actual error trace. After the refinement procedure, the tree is reconstructed, because changes of assertions may also change the shape of the tree. Given the new tree, BLAST starts a new iteration of the procedure we described in this paragraph so far.

The heart of the refinement routine is its ability to make the current abstraction more precise. It is implemented in BLAST by Craig's interpolation [7]. $\psi$ is a Craig interpolant of two formulas $\varphi^-$and of $\varphi^+$, if the following conditions are satisfied.

1. $\varphi^- \wedge \varphi^+$ is unsatisfiable
2. $\varphi^- \Rightarrow \psi$
3. $\psi \wedge \varphi^+$ is unsatisfiable
4. $\psi$ only contains symbols common to $\varphi^-$ and $\varphi^+$

Given an appropriate logic theory, such interpolants always exist. BLAST follows the error trace step by step, by adding the predicates found on the arrows

[8] In this context, abstraction denotes all assertion on program variables. The term is appropriate, because it expresses the fact that the current assertion might not be as precise as possible.

to $\varphi^-$, the so called path formula. $\varphi^+$ denotes the predicates on the rest of the path to the error label. In order to respect the effect of assignments, BLAST uses the single assignment form[9] . At every state, BLAST applies Craig interpolation, if $\varphi^- \wedge \varphi^+$ is unsatisfiable. The result of the Craig interpolation $\psi$ is added as new assertion on the state, where contradiction was found.

**Example** Figure 10 contains the ART, when a first path to the error state was discovered. Two state annotations can be found by Craig interpolation.

**Annotation for the loop-condition State** If we follow the error trace (bold arrows) in figure 10, we find at the first state that $\varphi^-$ is $i \geq 0$ and $\varphi^+$ is $i \leq 0 \wedge i \neq 0$. The Craig interpolant $\psi$ is $i \geq 0$, because

1. $i \geq 0 \Rightarrow i \geq 0$
2. $i \geq 0 \wedge i \leq 0 \wedge i \neq 0$ is unsatisfiable
3. $\psi$ only contains symbols common to $\varphi^-$ and $\varphi^+$

**Annotation for the Postcond State** We follow the error trace, by taking the step from the loop-condition state towards the postcond state. We find that $\varphi^-$ is $i \geq 0 \wedge i \leq 0$ and $\varphi^+$ is $i \neq 0$. The Craig interpolant $\psi$ is $i = 0$, because

1. $i \geq 0 \wedge i \leq 0 \Rightarrow i = 0$
2. $i = 0 \wedge i \neq 0$ is unsatisfiable
3. $\psi$ only contains symbols common to $\varphi^-$ and $\varphi^+$

The current ART is modified by inserting the two assertion $\psi$ found. The error trace contains a contradiction, which means that the this path is not feasible. A new, refined ART (figure 11) is the result.

The error state is still reachable, by crossing the loop once. Again, by applying Craig interpolation, we find a predicate, allowing us to say that the situation after the loop is equivalent to the situation before the loop. This observation leads to the final ART in figure 12.

BLAST refines iteratively the ART. Two [10] events may stop that process. Either a reachability of the error state can be excluded, or a feasible trace to the error state is found.

[9] An assignment can change the value of a variable $x$, and predicates concerning $x$ before and beyond the assignment don't refer to the same $x$. Therefore, we give an index to $x$, which is changed everytime something is assigned to x. A path formula like $x > 0 \wedge x = x - 1 \wedge x = 0$, would be written as $x_1 > 0 \wedge x_2 = x_1 - 1 \wedge x_2 = 0$, in single assignment form.

[10] Technically, there exist a third option. BLAST may also terminate because it does not find new predicates.

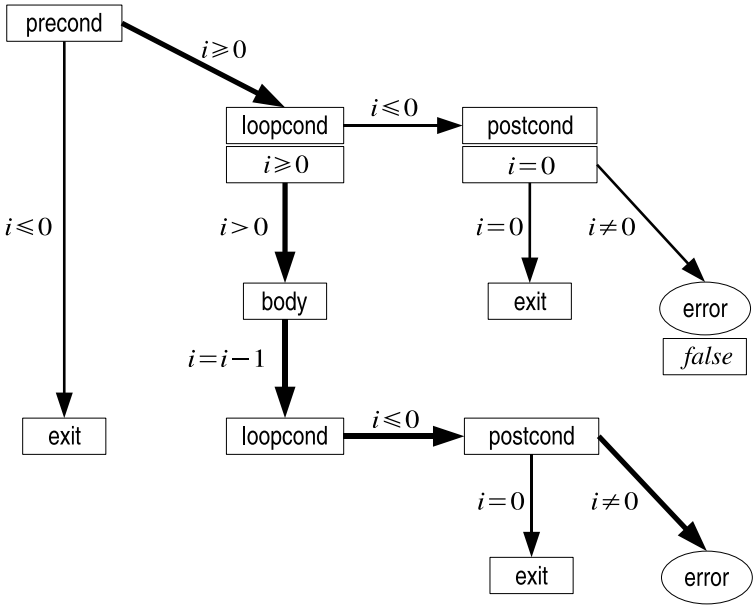


Figure 11: Second ART of the simple decrement problem.

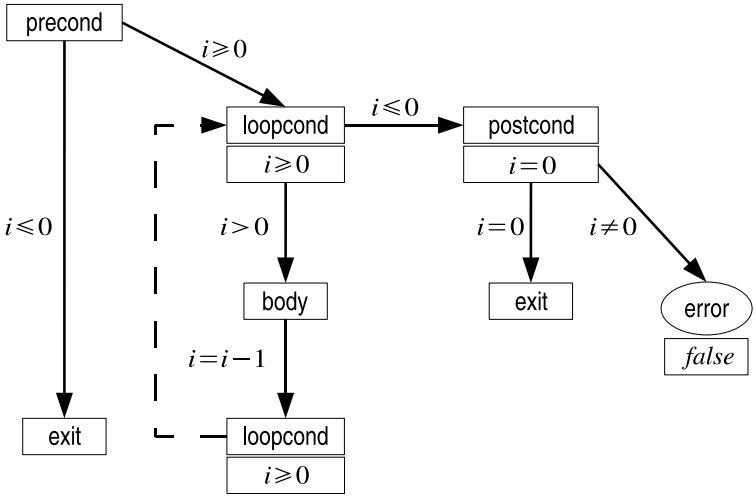


Figure 12: Final ART of the simple decrement problem.

## 2.4 Partial and total correctness

Two levels of correctness are usually distinguished in formal verification, partial and total correctness. Partial correctness consists in demanding, that the post-condition is never violated. Total correctness additionally imposes the program to terminate. If BLAST shows that an error state is not reachable, this doesn't tell anything why this is the case. It is possible that just before the error state an infinite loop blocks the program execution. Therefore, the postcondition is just a safety condition in the context of model checkers. That's why we speak of partial correctness, meanwhile we denote by complete correctness, if termination is shown by theorem proving.

Dynamic logic encodes total correctness by the diamond operator (see section 2.2.3), partial correctness by the box operator "[{ }]".

# 3 Discovering&Exploiting Invariants

## 3.1 Overview

The advantage of the model checker BLAST is the fact, that no user interaction is necessary when proving. KeY, an interactive theorem prover, provides automated heuristics, but they usually fail when working with loops.

We start BLAST first, if successful we look for invariants and feed the KeY prover with them. This approach can also be applied to subgoals in a proof. Invariants can be applied in the KeY system, using a special proof rule. In contrast to BLAST, we can prove total correctness in KeY. In this way, we combine the power of KeY with the comfort of BLAST.

Please note that in the following, we will sometimes use the abbreviation conventions in figure 13.

| Symbol | Meaning |
|---|---|
| $\phi$ | precondition |
| $\psi$ | postcondition |
| $\gamma$ | loop-condition |
| $\omega$ | invariant |
| $\chi$ | variant |

Figure 13: Abbreviation symbol table.

## 3.2 Loop Invariants

A loop invariant is basically an assertion on the program variables, being true when the condition of the loop is evaluated. During the execution of the body, the invariant is not supposed to be true. The invariant assertion is an important information for the formal proof of a program containing a loop. In fact, we say it is strong enough, if it is possible to show correctness of the postcondition by combining the invariant with the negation of the loop-condition. The simplest invariant is the assertion *true*, but in almost all cases, this is not enough to show the postcondition. Hence, the challenge is to discover invariants being strong enough.

## 3.3 Problem Blueprint

To keep the ideas simple, we define the problem to solve. We consider problems containing one loop. Nested loops or sequences of loops are not allowed so far. A problem blueprint for KeY and BLAST can be found at figure 14.

## 3.4 Discovering Invariants in ARTs

This chapter is dedicated to the art of finding invariants in ARTs. We focus here on problems respecting the defined blueprint. A precondition is imposed,

```
if(precondition)
{
   while(loopcondition)
   {
      body
   }
   if (!(postcondition))
   {
      ERROR:
   }
}
```

BLAST style

```
precondition
->
{updates}
\<{
   while(loopcondition)
   {
      body
   }
}\>
postcondition
```

KeY style

Figure 14: Problem blueprint in BLAST and in KeY style. No other loop is contained in the body.

and after the execution of the loop, a postcondition is demanded. Using an example (see figure 15), we present our idea on how we find the invariant. The corresponding ART can be found at figure 16. We simplified the ART diagram, such that there are only states important for the control flow.

In order to be sure that no path leads to the error, BLAST generates all possible traces through the body of the loop. If we look at the ART, we can see that first the preconditions are processed. If they are violated, we do not say anything on the program and its postcondition. In the contrary case, we enter the loop a first time. It is not possible that we don't even enter once, because by the precondition we know that $z1 > 0$, initially. Then, $z1$ and $z2$ are decremented until one of them is zero. If $z1$ is zero, we exit the loop and BLAST guarantees, that the postcondition is satisfied. If $z2$ is zero, we go on and decrement $z3$ to zero. Afterwards, $z1$ is finally decremented to zero, and the program terminates. An interesting point is, that BLAST does not encode in the ART that $z3$ is decremented first, and $z1$ afterwards. Although this is the case in the real program, BLAST is too lazy to check that out. In fact, it guarantees already at this abstraction level, the postcondition is never violated. We stated earlier, that an invariant should

1. always be true before the loop-condition is checked.
2. be strong enough to prove the postcondition.

If we look at the sample ART, we can see that a candidate for the invariant must be the expression $\alpha_0 \vee \alpha_1 \vee \alpha_2 \vee \alpha_3$. It satisfies the first invariant criterion, because BLAST generates all possible traces at a certain level of abstraction. Further, we know that the $\alpha_i$ are true at the corresponding state on the trace. Because

**Precondition:** $z1 > 0 \wedge z2 \geq 0 \wedge z3 > 0$

```
while (z1 > 0)
{
   if (z2 > 0)
   {
      z1--;
      z2--;
   }
   else if (z3 > 0)
   {
      z3--;
   }
   else
   {
      z1--;
   }
}
```

**Postcondition:** $z1 = 0 \wedge (z3 > 0 \vee z3 = 0)$

Figure 15: Example problem for invariant discovery.

we connect the $\alpha_i$ by an or operator, we can conclude that every time before the loop-condition is evaluated, one of the $\alpha_i$ is true. The second invariant criterion is satisfied under the assumption, the annotation of a state resumes all important information so far. We discuss this in detail in section 3.7.1, here we assume it is true. A good way to understand is to go backwards on the trace of the ART. Let's start at the postcondition evaluation. Here, BLAST can guarantee that the postcondition is true, otherwise the refinement would not have stopped or an error had been found. By going a step backwards, we see that the negation of the loop-condition $\neg\gamma$ has been added to the path, before the postcondition check. Now we are for sure at a state before the loop-condition step, because our problem blueprint does not allow to have other program statements behind the loop. We can see, that the state is annotated by $\alpha_i$, hence $\alpha_i \wedge \neg\gamma$ is strong enough to show the postcondition is true (remember the assumption, that all important information is resumed in an annotation). We should also have a look at the case of $\alpha_0$, at the beginning of the trace. The postcondition is out of reach. This is not a problem, $\alpha_0 \wedge \neg\gamma$ is strong enough too, because it is contradictory and does imply anything by definition.

In general, we state the invariant is $\alpha_0 \vee ... \vee \alpha_n$, if we have n loop-condition states and if we denote their assertion by $\alpha_i$. Loop-condition states that are leafs can be ignored, because their assertions are contained by definition in internal nodes. In section 3.7, we discuss why this invariant is strong enough and fulfills all necessary formal criteria.

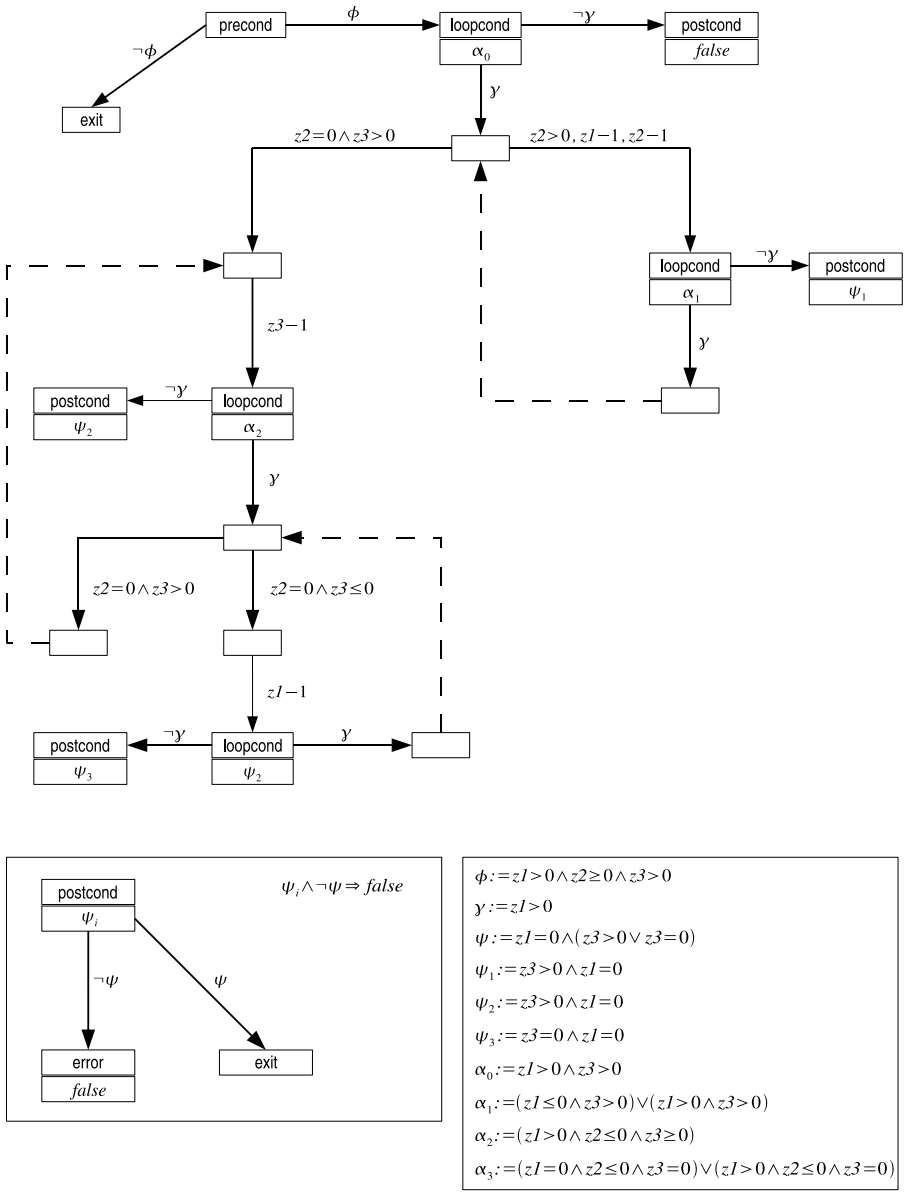


Figure 16: This is the simplified ART of the example problem. $\phi$ is the pre-condition, $\gamma$ the loop-condition and $\psi$ the postcondition. The $\alpha_i$ denote the conditions true before the loop-condition is evaluated. We use shorthand for the postcondition part of the ART, detailed in the left legend.

$\phi(n_0, .., n_L)$
$\Rightarrow$
$\{m_0 := f_0(n_0, .., n_L)\} .. \{m_P := f_P(n_0, .., n_L)\}$
$< \{while(\gamma(m_0, .., m_P))\{body\}\} >$
$\psi(m_0, .., m_P, n_0, .., n_L)$

Figure 17: Problem blueprint in formal dynamic logic. We assume, that no other while loop is situated in the body.

## 3.5 Variant

The variant is a function of the program variables, having the following properties.

1. At each iteration step of the loop, the variant gets smaller.
2. If the variant is smaller or equal to a fixed $n \in \mathbb{Z}$, the loop-condition evaluates to false.

The first property ensures, that the execution of the loop does never freeze. The second property ensures the existence of an interval $I = (-\infty, n]$; $n \in \mathbb{Z}$, where the loop-condition is false, if the variant is in $I$. Both properties together guarantee termination, because a strictly decreasing function reaches such an interval $I$ necessarily.

For the moment, we leave the discovery of the variant as an unsolved problem to the user. Note that the BLAST proof does not contain information that could yield the variant. This is because BLAST can ensure, that the postcondition is never violated, but it does not tell something about termination explicitly. We conclude, that an invariant always can be found in the ART, because this concept is related to the correctness of the postcondition. Further, we conclude that the variant is not necessarily in the ART, because the model checking approach of BLAST does not cover termination.

## 3.6 The Invariant Taclet

KeY contains a taclet that allows to prove while loop programs by using the invariant [4]. It's our interface to use the information from the BLAST proof, within KeY.

The invariant taclet is well appropriated for our approach. The tasks of proving termination and correctness of the postcondition are separated, by the concepts of the variant and invariant.

The problem blueprint from figure 14 would look like the statement shown in figure 17 when reformulating it in dynamic logic. We assume the problem has L logical variables, denoted by $n_0$ up to $n_L$. Additionally, we assume there are P program variables, denoted by $m_0$ up to $m_P$. The program variables are initialized by functions of the logical variables, denoted by $f_0$ upto $f_P$.

| | |
|---|---|
| invariant initially valid | $\vdash \phi \Rightarrow \omega$ |
| body preserves invariant | $\vdash \omega \Rightarrow (\gamma \Rightarrow [body]\omega)$ |
| variant decreasing | $\vdash \omega \wedge \chi > 0 \Rightarrow \gamma \Rightarrow < body > (\chi < \chi@pre)$ |
| termination | $\vdash \omega \wedge \chi \leq 0 \Rightarrow \neg\gamma$ |
| use case | $\vdash \omega \wedge \neg\gamma \Rightarrow \psi$ |

Figure 18: The five sub-goals of the while invariant taclet in KeY.

A problem of the form as defined in figure 17 can be solved by the invariant taclet of KeY (see figure 18). Basically, the user of KeY must deliver two informations for that proof rule. First, there is the invariant, we discussed already in the previous sections. As the second formula, the taclet needs is the variant, denoted by $\chi$ in this context.

In the next five sections, we explain the sub-goals of the invariant taclet, introduced in figure 18.

#### 3.6.1 Invariant Initially Valid

$$\vdash \phi \Rightarrow \omega$$

The goal demands, that a given invariant $\omega$ is valid, when the loop is entered the first time. Logically this means, that the invariant is a consequence of the precondition $\phi$.

#### 3.6.2 Body Preserves Invariant

$$\vdash \omega \Rightarrow (\gamma \Rightarrow [body]\omega)$$

This goal exists, because it guarantees that $\omega$ is a real invariant. We assume the invariant $\omega$ and the loop-condition $\gamma$ are true. The goal demands, given the assumptions, that if the body is executed, $\omega$ remains true.

#### 3.6.3 Variant Decreasing

$$\vdash \omega \wedge \chi > 0 \Rightarrow \gamma \Rightarrow < body > (\chi < \chi@pre)$$

The goal here is to ensure, the variant decreases at each iteration. In other words we prove, that by every possible execution of the loop body, we do a step towards the termination of the loop.

#### 3.6.4 Termination

$$\vdash \omega \wedge \chi \leq 0 \Rightarrow \neg\gamma$$

To show termination we are supposed to prove, given the invariant is true and the variant smaller or equal to zero, the loop-condition is false. We stated in section 3.5, that $\chi$ must be smaller or equal than an arbitrary $n \in \mathbb{Z}$, but

here the taclet imposes $n$ to be zero. This is not a principal problem, because we can transform a variant $\chi_n$ suitable for $n$ to a variant $\chi_0$ suitable for 0 by $\chi_0 = \chi_n - n$.

### 3.6.5 Use Case

$$\vdash \omega \wedge \neg\gamma \Rightarrow \psi$$

This claim goal ensures the invariant to be strong enough to show the postcondition. As already mentioned at section 3.2, an invariant is strong enough, if in combination with the negation of the loop-condition, it can be used to show the postcondition.

## 3.7 BLAST's Invariant in KeY

We show in this section, why an invariant of the form of section 3.4 can be applied successfully using the taclet of section 3.6. The application of the taclet creates five new sub-goals. For the simple examples we investigated, the goals are simple enough, such that the heuristics of the KeY system can solve them automatically.

### 3.7.1 Relation between State Annotations in ARTs

We introduce here an important property of state annotations in an ART, because it helps to understand the correctness of the BLAST invariant.

We start with an observation at an arbitrary ART state, annotated by $\alpha_i$. From here, we walk along one specific path, by adding the predicates and updates $p_i$ to the path formula. The formula is constructed using the single assignment form (see section 2.3.3). After n steps, we arrive at a state annotated by $\alpha_j$ (see figure 19). BLAST implements Craig interpolation in a way, such the following statement is true.

$$\alpha_i^{path} \wedge p_0 \wedge \ .. \ \wedge p_n \Rightarrow \alpha_j^{path}$$

The *path*-index means, that the $\alpha$-statements are written by variables indexed by the single assignment procedure. In BLAST, the Craig interpolation produces results with indexed variables such as $x_1 < 4 \wedge y_2 = 4$, but the final state annotation is $x < 4 \wedge y = 4$. We distinguish this two notations by the *path*-index, such that we can write the property properly.

### 3.7.2 Equivalence of Symbolic Program Execution and the Path Formula

We can combine the model checker and the theorem prover paradigm, because of the equivalence between path formula and symbolic execution. The implication $\varphi^- \Rightarrow \psi$ of the Craig interpolation can be used by the theorem prover. If the theorem prover executes symbolically the path represented by $\varphi^-$, we know

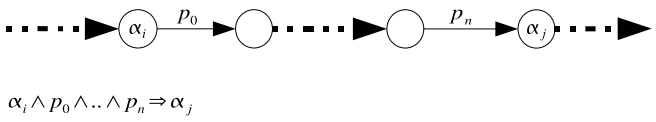


Figure 19: The annotation of a state resumes all important information so far.

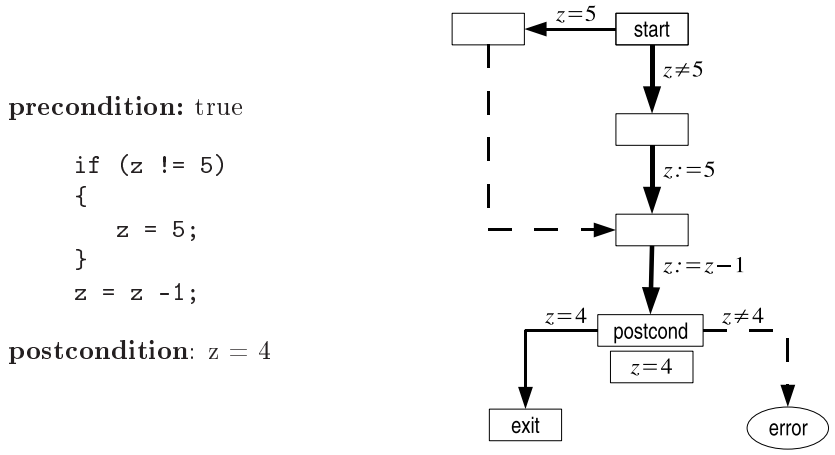


Figure 20: Toy problem and its ART.

that $\psi$ can be concluded. An assignment in symbolic execution changes the update values of the variable on the left hand side of the assignment. This allows the prover to keep track of the actual value of the program variable in the logic context. The same effect has the single assignment policy. For every assignment, a new instance of the variable is introduced, representing the actual value. A predicate on the path is introduced directly with the actual instances of the variables in BLAST. In KeY, the predicate is added to the hypotheses using the actual update values.

The following example may help to fix the idea. Let's have a look at the toy problem and its ART in figure 20. We are interested in the bold trace of the ART. The trace formula up to the postcondition state is

$$z_0 \neq 5 \wedge z_1 = 5 \wedge z_2 = z_1 - 1 \Rightarrow z_2 = 4$$

The annotation $z = 4$ is sufficient to conclude the program is safe. We demonstrate now, how BLASTs trace formula can be found in the equivalent KeY proof. Initially, we assume that $z$ is equal to an arbitrary $z_0$.

```
==>
{z := z_0} <{ if (z != 5) {z = 5;} z = z - 1; }> z = 4
```

The first rule application concerns the if-statement. We split the proof into two sub-goals, corresponding to the possibilities that $z \neq 5$ and $z = 5$. The same fact is represented by the two outgoing arrows from the start-state in figure 20. We consider here the case $z \neq 5$, because it corresponds to trace we have chosen for the path formula.

$z_0 \neq 5$
==>
{z := $z_0$} <{ z = 5; z = z - 1; }> z = 4

The second step treats the assignment $z = 5$. KeY would by default change the update directly to $\{z := 5\}$. We use a less direct procedure for our demonstration. We introduce an intermediate logical variable $z_1$.

$z_0 \neq 5,\ z_1 = 5$
==>
{z := $z_1$} <{ z = z - 1; }> z = 4

Instead of assigning the value 5 directly to $z$, we use the new variable $z_1$. In the hypotheses, we specify $z_1 = 5$. This ensures that we do exactly the same as KeY does originally. We proceed in the same way with the next assignment.

$z_0 \neq 5,\ z_1 = 5,\ z_2 = z_1 - 1$
==>
{z := $z_2$} <{ }> z = 4

The diamond is empty, because the program has been executed completely. Therefore, we can remove the diamond and assign the actual update values to the program variables to the postcondition.

$z_0 \neq 5,\ z_1 = 5,\ z_2 = z_1 - 1$
==>
$z_2$ = 4

At this stage, the remaining goal is equivalent to the path formula, because we introduced variables to fix the update values. We conclude that a state annotation somewhere on a trace is also true at the corresponding moment at symbolic program execution.

### 3.7.3 Invariant Discussion

We claimed in section 3.4, that $\alpha_0 \vee \ ... \ \vee \alpha_n$ is a valid invariant for a problem respecting the blueprint in figure 17. In this section, we show that the invariant does fulfill the formal requirements of the *while invariant* taclet. We discuss for that reason the three sub-goals, concerning the invariant. Formally, we replace $\omega$, representing a general invariant in the taclet, by our invariant $\alpha_0 \vee \ ... \ \vee \alpha_n$. We can show by using the properties introduced in sections 3.7.1 and 3.7.2, that our invariant fulfills the taclet's requirements.

**Invariant Initially Valid**

$$\Gamma, \phi \vdash \alpha_0 \vee \ ... \ \vee \alpha_n$$

The invariant is supposed to be true, given the preconditions as hypothesis. Let's denote the assertions of the ART nodes, corresponding to the moment we enter the loop the first time, by $\alpha_{init}$. From Craig interpolation we know, that

$$\varphi^- \Rightarrow \psi,$$

given $\varphi^-$ is the path up to a state and $\psi$ is the assertion on the state. From this observation, we can conclude that

$$\phi \Rightarrow \alpha_{init}.$$

This reflects the fact, that a path across the preconditions leading to one of the $\alpha_{init}$ exists, for every initial loop-condition state.

**Body Preserves Invariant**

$$\Gamma, \alpha_0 \vee \ ... \ \vee \alpha_n, \gamma \vdash [body]\alpha_0 \vee \ ... \ \vee \alpha_n$$

By construction of the invariant, we are supposed to show preservation, for each $\alpha_i$ given as hypothesis. More formal, the proof of the goal above is equivalent to the proof of $n$ sub-goals, of the form

$$\Gamma, \alpha_i, \gamma \vdash [body]\alpha_0 \vee \ ... \ \vee \alpha_n.$$

In order to be more precise, we have to mention the updates preceding the [*body*]-statement. We denote in the following the variables modified in [*body*] by $m_j^{modi}$, the others by $m_j^{const}$. In order to represent an arbitrary situation of loop execution, the taclet introduces a new logical variable $n_j^{new}$ for each modified variable. The sub-goal above can be written as the following statement, by introducing that notation (remember also, that $f_j$ is the original initialization of the program variable $m_j$).

$$\Gamma, \alpha_i[m_j^{modi}/n_j^{new}], \gamma[m_j^{modi}/n_j^{new}] \vdash$$
$$\{m_j^{modi} := n_j^{new}\} \,..\, \{m_j^{const} := f_j(n_0, \,..\,, n_L)\}[body]\alpha_0 \vee \ ... \ \vee \alpha_n$$

All variables modified in [*body*] have to be initialized by a new logical variable. Because the invariant and the loop-condition $\gamma$ is specified in terms of program variables, we also replace the occurrences of the program variables by the corresponding logical variables in these terms. The aim of the replacements and the update modifications is to guarantee, that we are in an arbitrary iteration of the loop's execution.

We explain in the following, why a sub-goal of this form is true. Let's revisit for that purpose the example of section 3.4. We show again the same ART here, but we use another layout to point out the idea (see figure 21). The states

are grouped in three zones. One zone represents the body of the loop, one the loop-conditions and one the postcondition checks. The loopbody zone may be very complicated, and there may be much more loop-condition states, but the zones still can be identified. Figure 22 resumes the three possibilities on what may happen to a trace entering the loop-condition body.

**Trace crosses Body** The ART trace outgoing from the loopcond-zone towards the loopbody-zone comes back to the loopcond-zone (left and right diagram on figure 22). Let's start an observation on state annotated by $\alpha_i$. We follow a trace through the loop body, by building the path formula $p_{i\to j}$[11]. Finally, we arrive at $\alpha_j$. By the property introduced in section 3.7.1, we know that $\alpha_i^{path} \wedge p_{i\to j} \Rightarrow \alpha_j^{path}$. This implication is important, because we know that we can make a link between the path formula and the symbolic execution (section 3.7.2). $\alpha_i^{path}$ corresponds to the hypothesis $\alpha_i$ in the goal. The path formula $p_{i\to j}$ is equivalent to what happens when the corresponding trace in [*body*] is unrolled. The Craig interpolation guarantees by the implicated $\alpha_j^{path}$, that one of the invariant's $\alpha_0, \ldots, \alpha_n$ is a valid postcondition for the sub-goal.

**Trace is Contradictory in Body** Further, we discuss the possibility shown in the central diagram of figure 22. The trace doesn't cross the body, because it's contradictory. Because of the equivalence between path formula and the symbolic execution, we can conclude that unrolling this trace leads to a contradiction in the hypotheses. A proof goal having contradictory hypotheses is true by definition.

**Use Case**

$$\Gamma \vdash (\alpha_0 \vee \ \ldots \ \vee \alpha_n) \wedge \neg\gamma \Rightarrow \psi$$

Because the invariant consists of several sub-terms connected by $\vee$-operators, we have to prove in fact n sub-goals of the form

$$\Gamma \vdash \alpha_i \wedge \neg\gamma \Rightarrow \psi.$$

In other words, we are supposed to show that each $\alpha_i$ is strong enough for the postcondition. The property, we introduced in section 3.7.1 helps us here. On figure 23 at the left hand side, the situation is outlined. If we quit the loop, we follow a trace annotated by $\neg\gamma$ [12]. Because we know that $\alpha_i^{path} \wedge p_{\neg\gamma} \Rightarrow \psi_i$[13]is true, we conclude the sub-goal is true by the equivalence of the paradigms (section 3.7.2). $\psi_i$ is by definition contradictory to $\neg\psi$ and does therefore not allow a violation of the postcondition.

[11] By $p_{i\to j}$, we denote the sequence of steps $p_0 \wedge \ldots \wedge p_n$, leading from the loop-condition state annotated by $\alpha_i$ to the one annotated by $\alpha_j$.

[12] If the loop-condition is not atomic, the path formula from $\alpha_i$ to the state before the error state implies $\neg\gamma$ by definition.

[13] We denote the subsequence of the path-formula from $\alpha_i$ upto the state before the error label by $p_{\neg\gamma}$.

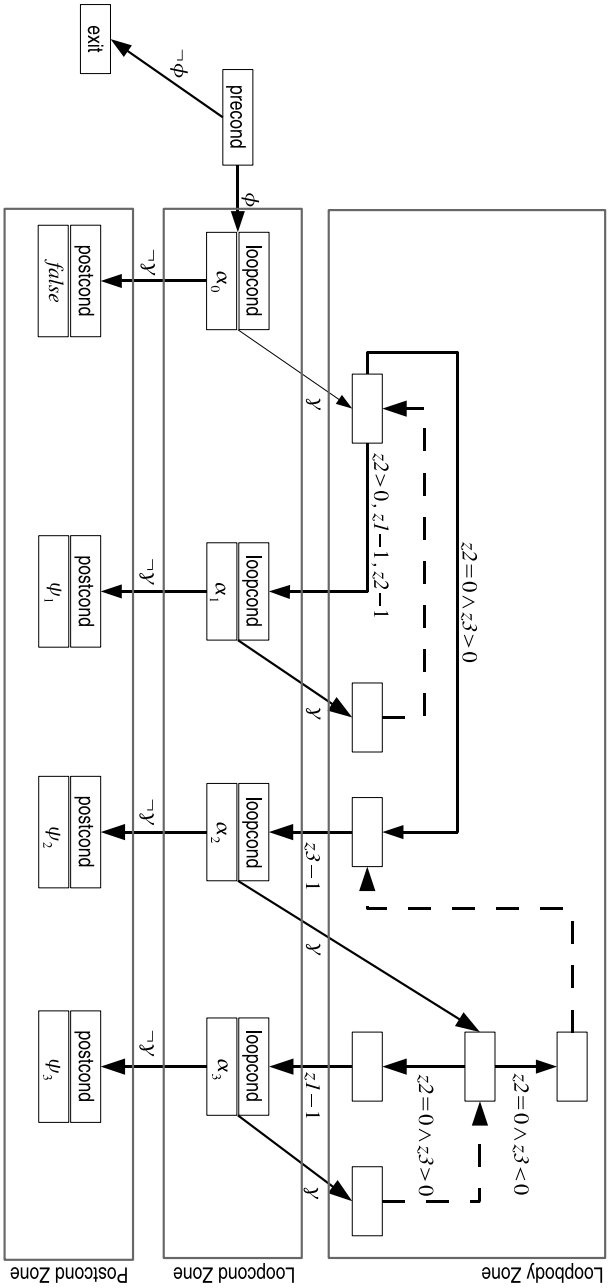


Figure 21: ART of figure 16, using another layout. We use the same conventions here as for the mentioned figure.

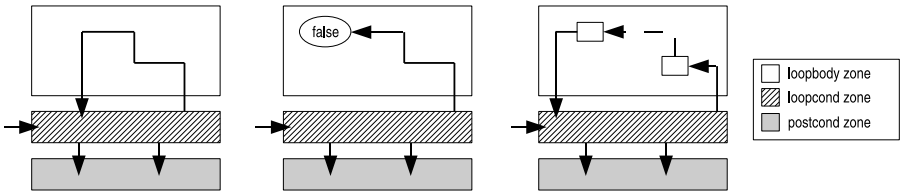


Figure 22: A trace into the loopbody zone matches to one of these three cases.

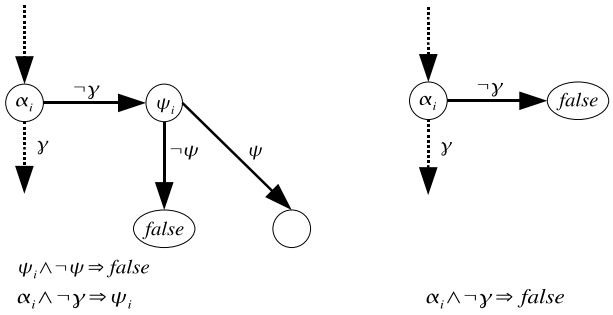


Figure 23: The invariant is strong enough, because each $\alpha_i$ is strong enough.

The conclusion remains true, if the postcondition isn't annotated by $\psi_i$, but by $false$. $\alpha_i^{path} \wedge p_{\neg\gamma} \Rightarrow false$ corresponds to a contradiction in the hypotheses of the sub-goal (see figure 23, righthand side).

### 3.7.4 Extension of the Blueprint

Up to here, we constrained ourselves to the given problem blueprint. In the current section, we discuss possible extensions.

**Code before the Loop** We assume here a problem, where some loop-free program code is executed, before we enter the loop. Figure 24 gives an example of such a program. An approach leading to success in such a case is symbolic execution of the piece of code before the loop. Because of the case distinction, we get two claims in the example case (see figure 25) . The code executed so far does influence the preconditions virtually. Because we assign $n1$ or $n2$ to $z1$, we enrich the preconditions by $z1 = n1$ for one case, and by $z1 = n2$ for the other case. Having done so, the problems to solve are of the known form, and we can apply the knowledge of the previous sections.

**precondition:** $n1 \geq 0 \wedge n2 \geq 0$

```
if (n1>n2)
{
   z1 = n1;
}
else
{
   z1 = n2;
}
while (z1 > 0)
{
   z1 = z1 - 1;
}
```

**postcondition:** $z1 = 0$

Figure 24: Loop-free program code before the while loop.

**pre:** $n1 \geq 0 \wedge n2 \geq 0 \wedge z1 = n1$

```
while (z1 > 0)
{
   z1 = z1 - 1;
}
```

**post:** $z1 = 0$

**pre:** $n1 \geq 0 \wedge n2 \geq 0 \wedge z1 = n2$

```
while (z1 > 0)
{
   z1 = z1 - 1;
}
```

**post:** $z1 = 0$

Figure 25: After symbolic execution, two claims in a simple form remain.

**Code after the Loop** The second extension concerns loop-free code after the while construct. First of all, we know that the invariant discovered remains a valid invariant, the code behind the loop does not influence it. Therefore, we don't have problems for the sub-claims of the invariant taclet, concerning termination and correctness of the invariant. More interesting is the use case, because the application of the taclet does result in a more difficult expression here. If we formalize the blueprint as in figure 26 on the left, the application of the while invariant taclet delivers a use case claim as shown on the same figure on the right. Such an expression does not impose any special problem.

**precondition:** $\phi$

```
while (loopcond)
{
   body
}
postcode
```

$\Gamma \vdash \omega \wedge \neg\gamma \Rightarrow < postcode > \psi$

**postcondition:** $\psi$

| | |
|---|---|
| Extended Blueprint, containing code after the loop. | New use case claim. The postcondition must be true, after the postcode execution. |

Figure 26: Extended blueprint and use case claim.

**Loop Sequences** At first step towards the solution of loop sequences is already given in the previous paragraph. $\Gamma \vdash \omega \wedge \neg\gamma \Rightarrow < postcode > \psi$ is the use case of such a problem, if we denote the instructions after loop by *postcode*. We can apply the taclet a second time, using the same approach on finding the invariant for the second loop. We do not need to apply BLAST again, all necessary information is already contained on the first ART. This is granted, because the invariant of the second loop must also contain the information of the first loop. We know that for sure, otherwise BLAST couldn't exclude the traces to the error state.

**Nested Loops** The nested loop problem is more difficult, because of the inner loop. The invariant we find is strong enough for the use case, by the same arguments we used for the classic case.

The heuristics of the problem prover fail, when proving invariant preservation of the body. The body contains itself a loop, so it cannot be unrolled simply. The problem occurs again when we prove the decreasing nature of the variant. An approach to solve the sub-goals is to get the invariant of the inside loop by the same mean as for the outside loop and to apply the while invariant taclet.

However, BLAST proofs of nested loops get complicated. Therefore we did not study this problem detailed.

# 4 Software Documentation

The plugin for KeY we implemented, using the ideas of the previous chapters, is documented here. First, we present the architecture of the system, in order to give an overview. We try to show all important steps, such that a complete picture of the software work-flow gets visible. Second, we will point out some interesting features of the implementation. The goal is to document how we used the classes that KeY already provides.

## 4.1 Architecture

### 4.1.1 Classes

We introduce and resume all classes of the plugin in this section. A structural diagram of the situation is given in figure 27. We show all public methods of the class, in most cases. Notice that this diagram is part of a bigger picture, because the software is embedded in the KeY system. We only show classes concerning our plugin. With one exception, all classes are part of the blastapplication package.

**ARTNode** This class extends *DefaultMutableTreeNode*, an element of the JAVA standard library. All operations we expect of a tree node are already implemented in *DefaultMutableTreeNode*. We add only ART specific attributes and methods. The tree itself is rooted in the class *DotFileInterpreter*. Every node can have an arbitrary number of children. The list of successors can be extended by the *add* method. The function *getConnectorLabelAt* gives the annotation of the ART transition toward a child.

**BlastApplicationRule** This is the main class of the plugin. It implements the interface *BuiltInRule*, that allows the programmer to create a super-rule. Another example of a *BuiltInRule* is *UpdateSimplification*, dealing with the updates of the program variables. Because of the interface specification, an *isApplicable* and an *apply* method must be provided. *isApplicable* has the task to decide if the rule is visible in the context menu (see figure 28) of the user. *apply* is called, when the user selects the menu entry of the rule.

The class contains an instance of itself. We add this instance to a LinkedList in the *ProblemInitializer*. We follows this procedure to subscribe the new rule in the system.

**BlastSyntaxer** The aim of the syntaxer is simple. BLAST prints out predicates in a certain form, using prefixed notation. Because we want to use them in KeY, this class implements a parser and translator for such expressions. The *convertCondition* method has a string parameter for the BLAST expression, and returns a string, containing the translation for KeY. As an example, the expression $And[, i == 0, Or[, j == 9, j == 0]]$ would be translated into $i = 0\&(j = 9|j = 0)$.

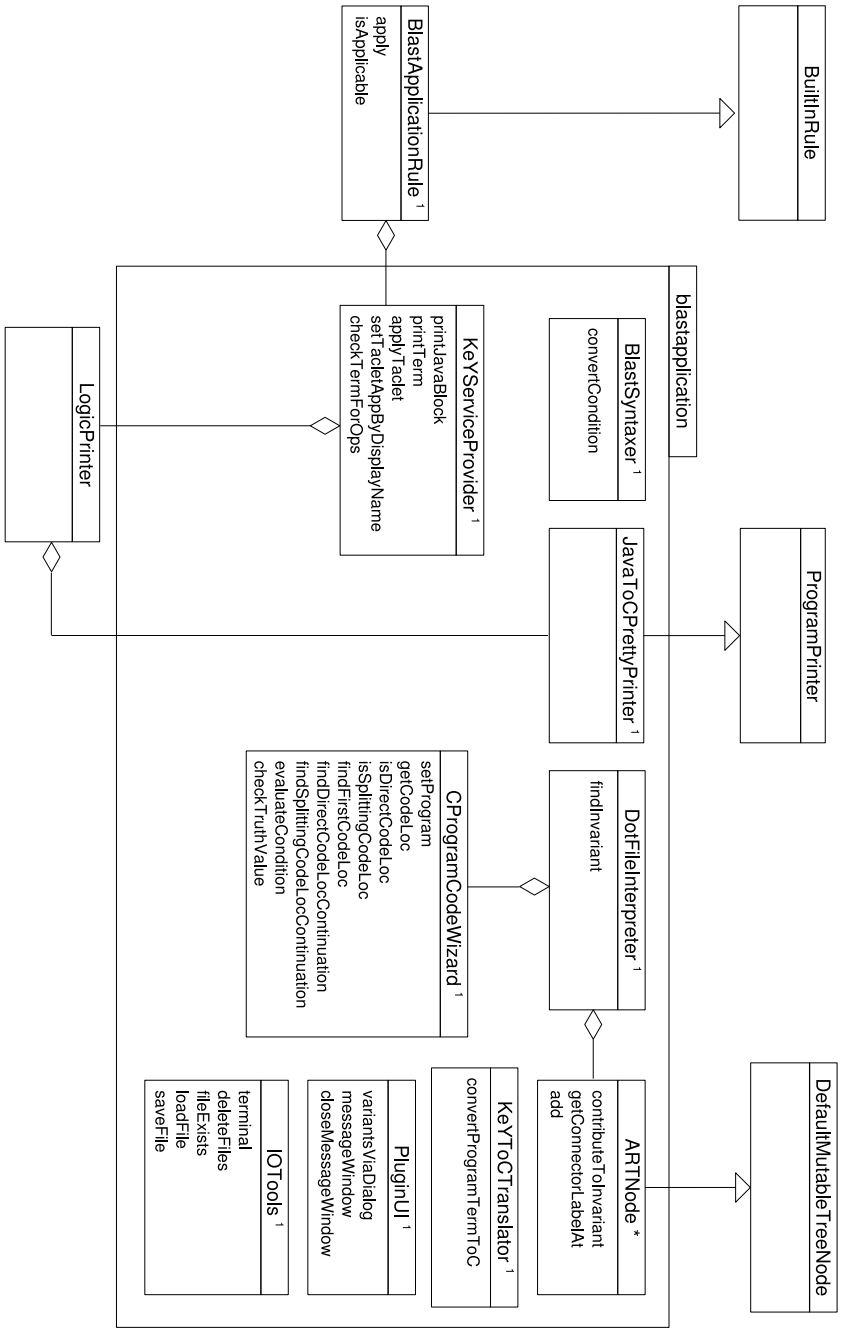


Figure 27: Class diagram of the plugin. We show the public methods of the classes, but we omit getters and setters.

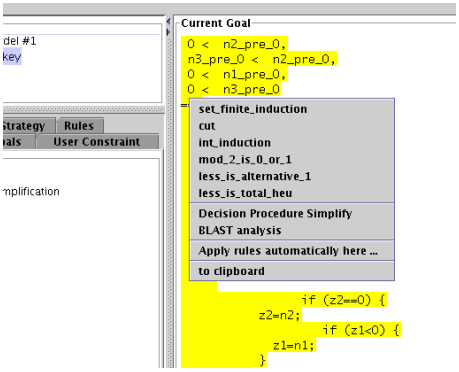


Figure 28: Context menu, extended by our rule BLAST analysis.

**CProgramCodeWizard** The aim of the wizard is to encapsulate functions working on C code. Basically, we want to annotate states of the ART by their location in the source code. This is not an easy task, because this implies code parsing, and the evaluation of conditions for C.

We explain the use of this class by the mean of pseudo code in figure 29. Remember the goal is to go through an ART and annotate the states.

We walk along the tree, by keeping track in the code. First of all, we set the root annotation to the result of the method *findFirstCodeLoc*. At each step, we test first if we are on a code location where the control flow is split.

- **Split Control Flow**. This is the case for *if* and *while* statements. A split control flow means, that the corresponding ART state has a child for the split-condition being true , and another for the split-condition being false. Because conditions of *while* and *if* statements can be composed by more than one atomic condition, the condition check may build a subtree within the ART. To keep track of the progress, we use the *condEvalProgress* construct, that stores the results of the atomic condition evaluations. If the progress is sufficient to show that the condition evaluates to true or to false, we look for the next code location. Otherwise, we remain on the same location, but with an extended *condEvalProgress*.

- **Direct Control Flow**. This case is simpler than the splitted control flow. We are in a situation, where the actual code location points on a series of simple statements. Because BLAST does handle them as a block in an ART, we jump behind that series here to find the new code location.

**DotFileInterpreter** If BLAST can solve a given problem, it generates a file called *reachtree.dot* containing the ART. The file is written using a standard notation, allowing to draw the tree automatically by the *dot* tool (which is part of the Graphviz toolset [9]). The interpreter class does only provide the

```
function start(ARTRoot,wizard)
{
   setLineNumbers
  (ARTRoot,wizard.findFirstCodeLoc,wizard,emptyProgress)
}
function setLineNumbers
(node,codeLoc,wizard,condEvalProgress)
{
   node.setCodeLoc(CodeLoc);
   if (wizard.isSplittingCodeLoc(codeLoc))
   {
      foreach child of node
      {
         p = extend condEvalProgress by true or false,
             depending on the child
         condEval = wizard.evaluateCondition(codeLoc,p)
         if (condEval is true or false)
         {
            newCodeLoc =
            wizard.findSplittingCodeLocContinuation
            (codeLoc,newCondEval)
            setLineNumbers
            (node.child,newCodeLoc,wizard,emptyProgress)
         }
         else
         {
            setLineNumbers(node.child,codeLoc,wizard,p)
         }
      }
   }
   else if (wizard.isDirectCodeLoc(codeLoc))
   {
      newCodeLoc = wizard.findDirectCodeLocContinuation
                   (codeLoc)
      setLineNumbers
     (node.child,newCodeLoc,wizard,emptyProgress)
   }
}
```

Figure 29: Pseudo code illustrating how to use the CProgramCodeWizard, to annotate an ART.

*findInvariant* method. If it is called, the ART is builded (using *ARTNodes*), on the base of the *reachtree.dot* file. By applying the *CProgramCodeWizard* as described in figure 29, the states of the ART are annotated. The reason of this step is, that we want to identify the states containing interesting annotations. The invariant is then picked of the tree, by implementing the idea explained in section 3.4.

**IOTools** This class is a toolbox for the dialog with the operating system. The goal is to simplify the program code, by adding a new level of abstraction.

- *terminal* takes a string, and executes it on the terminal. It interrupts the program flow, until the order has been processed.
- *deleteFiles* takes a list of files and deletes them, if they exist.
- *fileExists* returns true, if a given file exist, false otherwise.
- *loadFile* takes a filename as parameter, and returns a string containing the content.
- *saveFile* stores a given content to a given location.

**JavaToCPrettyPrinter** Key uses a pretty printer to display a JAVA program in a nice form. A program is stored as a tree internally. An instance of the printer class does visit each node of that tree. Depending on the type of the node, the corresponding method of the printer is called. JAVA and C have very similar syntax at the base. Because we treat basic programs so far, we did not change any syntax, but we forbade JAVA specific structures such as exceptions. For that purpose, we extended the dangerous methods of the *PrettyPrinter*, and throw an exception if called. We don't show the methods on the diagram, because there are too many.

**KeYServiceProvider** The service provider encapsulates the exchange with the KeY system. It simplifies the internal application of taclet for the programmer. An instance of the class can be created, by passing the goal to resolve in the constructor.

The method *setTacletAppByDisplayName* prepares the application of a taclet. It has one parameter of type string, that should contain the name of the taclet the programmer wants to apply. If such a taclet cannot be applied on the goal, the method returns false.

After setting up the taclet, *applyTaclet* can be called. Instantiations values for the taclet are transmitted via parameters of the method. We use a simple pattern matching, to connect the available input fields of the taclet with the instantiation. The method may throw an exception, when problems concerning the instantiation are encountered.

The *printTerm* and *printJavaBlock* method form a second group of methods. They make use of KeYs *LogicPrinter* class. We modify the logic printer in two ways.

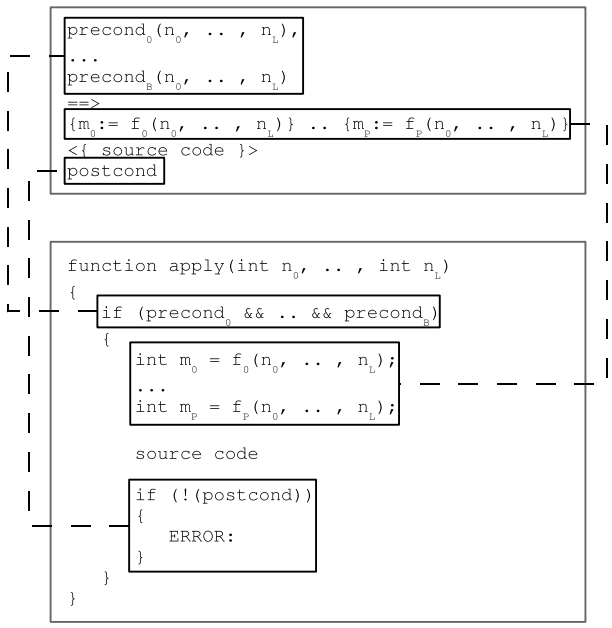


Figure 30: Idea of KeY (box above) to BLAST (box below) translation. We assume to have L logic variables $n_i$, B precondition terms, P program variables $m_i$ with a corresponding initialization function $f_i$.

1. Terms, formerly written in KeY syntax (ex. $a = 4|b >= 3$), can be printed using C syntax (ex. $a == 4 \,||\, b >= 3$).

2. JAVA programs can be translated to C, because of the *JavaToCPrettyPrinter*, our extension to the standard *PrettyPrinter*.

The methods allow to translate a given problem in KeY to a C program. This is not entirely implemented in this class, but we decided to provide the print methods in this class, because they make use of internal function of the KeY source code.

The *checkTermForOps* method finally checks if a term does only contain operators we allow. We filter the preconditions using this function, because BLAST cannot treat every sort of precondition possible in KeY.

**KeYToCTranslator** The purpose of the translator is to convert a KeY problem into a BLAST problem. The method *convertProgramTermToC* gets a *KeYServiceProvider* as parameter, in order to be able to print terms by using the classes of KeY. The main task of the translator is, to extract certain important pieces of the term and assemble them to a C program. Figure 30 shows the main steps.

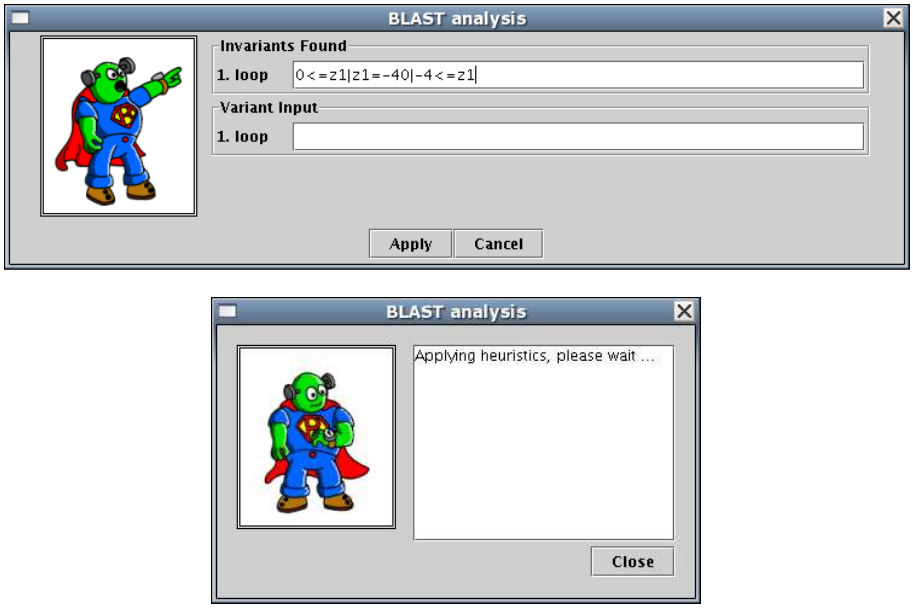


Figure 31: User input panels specified in the PluginUI class.

**PluginUI** We encapsulate the user interface in this specific class. The method *variantsViaDialog* takes a LinkedList containing the invariants discovered so far. It gives the user the possibility to modify the invariant and to specify a variant. If apply is pushed, the method returns the invariant and variant specified. The *messageWindow* method has two strings as parameter. This allows the programmer to communicate with the user, by specifying an image and a text. The programmer can close this window in the code by the *closeMessageWindow* method. Two samples of such windows are shown in figure 31.

### 4.1.2 Collaboration Diagrams

We present in this section the most interesting collaboration diagrams. We document by using them the work-flow of the most important actions. To increase the readability, we omitted function parameters on the diagram.

**Launch the Plugin by *apply*** The method *apply* in the class BlastApplicationRule is called by the interface, if the user starts our plugin. Most of the work to do is outsourced to other classes. Our goal of this dispatching is to increase code readability in the *BlastApplicationRule* class. The collaboration diagram of *apply* is in figure 32.

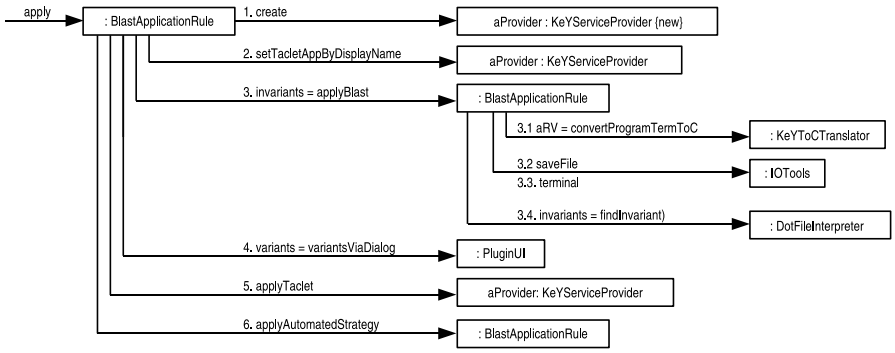


Figure 32: Collaboration diagram of the method *apply* in class *BlastApplicationRule.*

| | | |
|---|---|---|
| 1. | create (goal, services) | A service provider object is created. It stores all information concerning the proof. |
| 2. | setTacletAppByDisplayName ( "while_invariant_...") | This method call prepares the taclet application. The name of the taclet is passed by parameter. If the taclet cannot be applied, the method returns false. We assume the taclet can be applied. |
| 3. | applyBlast (aProvider) | This function groups the necessary operations to invoke BLAST. |
| 3.1 | aRV = convertProgramTermToC (aProvider) | The method returns a string, containing a C program. The parameter *aProvider* contains the current goal, which is the base for constructing the C source code. |
| 3.2 | saveFile (aRV, "blast.c") | The method *saveFile* of the *IOTools* class saves the content in the first parameter into a file named by the second parameter. |
| 3.3 | terminal ("pblast.opt blast.c ...") | The method executes BLAST in the terminal. We assume *pblast.opt* is in the path. The target file *blast.c* was written by operation 3.2. |

| | | |
|---|---|---|
| 3.4 | invariants = findInvariant ("reachtree.dot","blast.c") | The *findInvariant* method returns the invariant as a string. The parameters contain the names of the *dot* and the C file. The files have been generated at operation 3.2 and 3.3. |
| 4. | variants = variantsViaDialog (invariants) | The method pops up a window to the user. The user can specify the variant and control the invariant. |
| 5. | applyTaclet (patterns, instants) | Because we have the information on variant and invariant, the taclet set at operation 2. can be applied. |
| 6. | applyAutomatedStrategy | This function releases the automated heuristics of KeY. |

**Transform the Problem by *convertProgramTermToC*** This is the only public method of the *KeYToCTranslator*. It takes a *KeyServiceProvider* object as parameter, and returns a string with the corresponding C program for BLAST. Figure 33 contains the collaboration diagram of the method.

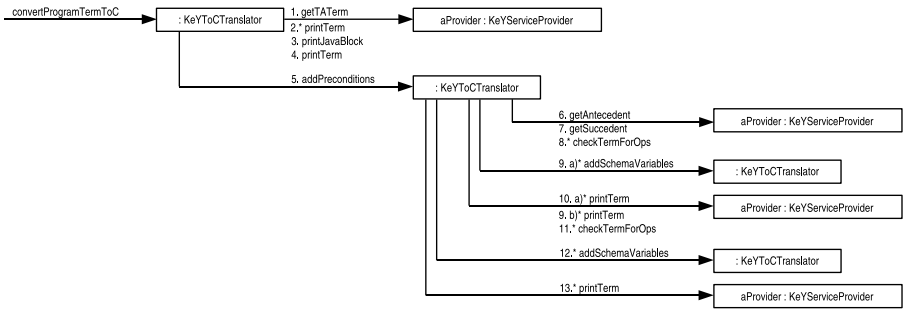


Figure 33: Collaboration diagram of method *convertProgramTermToC* from class *KeYToTranslator*.

| | | |
|---|---|---|
| 1. | getTATerm() | This getter returns the program term contained in the instance of the *KeYServiceProvider*. |
| 2. | printTerm (updateTerm, CStyle) | Print the updates in form of C integer variable declarations. |
| 3. | printJavaBlock(programTerm) | Print the JAVA program as C, using the extended pretty printer. |

| | | |
|---|---|---|
| 4. | printTerm(postTerm,CStyle) | Print the postcondition of the program in C style. |
| 5. | addPreconditions (aProvider, progString) | The functions adds the precondition tests to the *progString* created so far. |
| 6. | getAntecedent() | Get the antecedent, to find eventual preconditions. |
| 7. | getSuccedent() | Get the succedent, to find eventual preconditions. |
| 8. | checkTermForOps (antecedentFormula, validOps) | Check for each statement in the antecedent, if it contains just valid operators (and, or, ..). |
| 9. a) | addSchemaVariables (antecedentFormula) | If the formula contained only valid operators at 8., we add it to the schema variables. |
| 10. a) | printTerm (antecedentFormula, CStyle) | If the formula contained only valid operators at 8., we print it as a precondition into the C program. |
| 9. b) | printTerm (antecedentFormula,KeYStyle) | If the formula contained unsupported operators at 8., we print a message for the user in KeY syntax. |
| 11. | checkTermForOps (succedentFormula, validOps) | Check for each statement in the succedent, if it contains just valid operators (and, or, ..). |
| 12. | addSchemaVariables (succedentFormula) | If the formula contained only valid operators at 11., we add it to the schema variables. |
| 13. | printTerm (succedentFormula, CStyle) | If the formula contained only valid operators at 11., we print its negation as a precondition into the C program. |

**Invariant Discovery by *findInvariant*** This is an important method, because it encapsulates the discovery of the invariant in an ART. The ART is given under the form of a dot-file. This file format makes automated graph drawing possible by the *Graphviz* tool. The class *DotFileInterpreter* is able to read such a file into an internal tree, and to find the invariant. We implement here the theory developed above. The method calculates the code position of each ART state for that purpose, by using an instance of *CProgramCodeWizard*. In this way, we can locate the loop-condition states in the ART. The collaboration diagram of *findInvariant* is in figure 34.

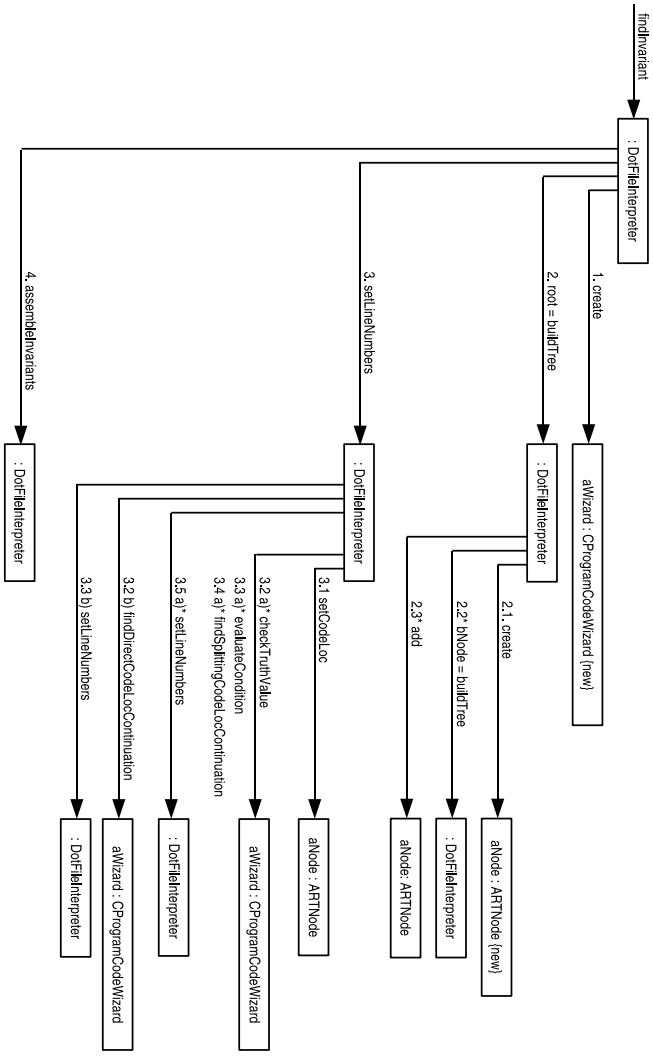


Figure 34: Collaboration diagram of method *findInvariant* in class *DotFileInterpreter*.

| | | |
|---|---|---|
| 1. | create(fileName) | We create an instance of the wizard, by specifying the name of the C file. |
| 2. | buildTree (firstNodeID, DotFileContent) | This method gets the content of the dot-file, and returns the root to the tree it builds, representing the ART. |

| | | |
|---|---|---|
| 2.1 | create(nodeID) | A node is created, for the node ID given by the dot-file. |
| 2.2 | buildTree (childID, DotFileContent) | For each child of the node in the ART, a child is created by a recursive call to *buildTree*. |
| 2.3 | add (childNode, arrowAnnoation, childID) | All children of the node are connected with the current node. |
| 3. | setLineNumbers (ARTRoot, firstCodeLoc, aWizard, emptyWay) | We start to calculate line numbers at the root. The way construct keeps track of the progress, when a complicated condition leads to several ART states. It stores somehow the way through the condition. |
| 3.1 | setCodeLoc(codeLoc) | The current ART state is annotated by the current code location. Depending on the type of the code location, we choose several times path a) or once path b). |
| 3.2 a) | checkTruthValue (arrowAnnotation) | The wizard checks, if the arrow to the current child-node represents the case the sub-conditions evaluates to true or to false. |
| 3.3 a) | evaluateCondition (codeLoc, aWay) | The wizard checks, if the current way through a condition is complete in the sense, that it evaluates to true or false. |
| 3.4 a) | findSplittingCodeLoc Continuation (codeLoc, conditionEval) | If the condition has evaluated to true or to false in 3.3 a), we go ahead in the code. |
| 3.5 a) | setLineNumbers (childNode, codeLoc, aWizard, aWay) | We call the function *setLineNumbers* recursively for the ART successor states. Either *codeLoc* or *aWay* has been modified by the current call. |
| 3.2 b) | findDirectCodeLoc Continuation (codeLoc) | The current ART state has only one successor. This means, the outgoing arrow represents a block of basic instructions in the program. The method finds the next position after the basic block in the program. |

| | | |
|---|---|---|
| 3.3 b) | setLineNumbers (childNode, codeLoc, aWizard, aWay) | We call the function *setLineNumbers* recursively for the successor state. |
| 4. | assembleInvariants (ARTRoot, whileCodeLoc) | The method walks through the tree and adds the annotation of loop-condition states to the invariant. |

## 4.2 Implementation Features

We resume in this chapter the most interesting points of the implementation. Someone who wants to extend KeY or our plugin, finds here the most interesting aspects we discovered or elaborated when implementing our plugin.

### 4.2.1 Integrate a Plugin into KeY

We integrated the plugin, by implementing *BuiltInRule*, an interface already given in the KeY source code. In this way, we created the class *BlastApplicationRule*. We oriented ourselves at the class *UpdateSimplificationRule*. *BlastApplicationRule* has two public methods, *apply* and *isApplicable*. The aim of *isApplicable* is to test whether the method can be applied on the current goal or not. If it returns $false$, the context menu does not contain the menu entry, allowing to apply the rule. The aim of *apply* is to start the execution of the rule, if chosen by the user in the context menu.

Finally, we have to register the new class to the KeY system. For that purpose, we add an instance of our class to a linked list in the *ProblemInitializer* class.

### 4.2.2 Taclet Application in the Source Code

First we have to create an instance of the class *TacletApp*, representing a taclet application. First, we create an iterator of the formulas in the succedent. We use the instruction

```
IteratorOfConstrainedFormula aItOfCF =
goal.node().sequent().succedent().iterator()
```

By a loop, we treat all elements of that iterator. Because an element is a formula, we want to know what taclets can be applied on that formula. We create another iterator of all possible *TacletApp*, by the following code.

```
bCF = (ConstrainedFormula)(aItOfCF.next());
aPosIC = new PosInOccurrence
(bCF, PosInTerm.TOP_LEVEL,goal.sequent());
aItOfTA = goal.ruleAppIndex().getTacletAppAt
(TacletFilter.TRUE,aPosIC,services,bCF.constraint())
.iterator();
```

We can identify the *TacletApps* by their names. In our case, we are looking for a *TacletApp* for the taclet called *while_invariant_with_variant_dec*. We test search by the instruction

```
if (tacletApp.rule().displayName().equals
("while_invariant_with_variant_dec"))
{
   myTacletApp = tacletApp;
}
```

The easiest way to instantiate the taclet, is to use the class *TacletInstantiationsTableModel*. The following lines of code demonstrate, how an instance of the class can be created.

```
NamespaceSet aNSPSet = goal.node().proof().getNamespaces();
AbbrevMap aMapOfAbr = goal.node().proof().abbreviations();
TacletInstantiationsTableModel aTableModel =
new TacletInstantiationsTableModel
(myTacletApp, services, aNSPSet, aMapOfAbr, goal);
```

The goal of this object creation is beeing able to instantiate the taclet as the user does, by simply writing a string into the correct field (see figure 35). If we want for example to instantiate a field called *variant* by the string *myVariant*, we can do it in the following way.

```
for (int i=0;i<aTableModel.getRowCount();i++)
{
   if (((SchemaVariable)(aTableModel.getValueAt(i,0))).
       toString().equals("variant"))
   {
      aTableModel.setValueAt(myVariant,i,1);
      break;
   }
}
```

We go through the rows of the internal table in the *TacletInstantiationsTableModel* by checking if a field of the first column contains the string *"variant"*. If we find such a row, we set the value of its second column to *myVariant*. By doing so for such field of the taclet, we can complete the instantiation. The following code lines can be used to apply the taclet.

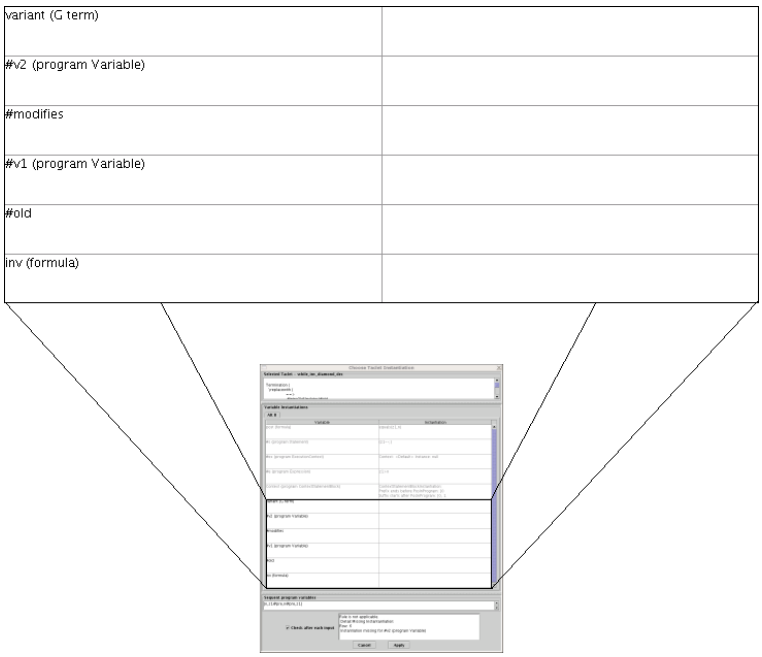


Figure 35: Taclet application window of the interface, for the *while_invariant* taclet.

```
aTA = aTableModel.createTacletAppFromVarInsts();
ListOfGoal result = goal.apply(aTA);
```

### 4.2.3 Heuristics and Simplification in the Source Code

It is possible to launch the heuristics and the simplifier in the source code. Additionally, the heuristics can be configured. In our code example, we set the number of steps to *numOfSteps* and the strategy on *"Simple JAVACardDL without unwinding loops and method bodies"*. The name of the strategy is stored in KeY as an instance of the JAVA default class Name. Instead of creating a new Name object, we use a getter of the *SimpleJavaCardDLOptions* class.

```
Main.getInstance(true).mediator().
setMaxAutomaticSteps(numOfSteps);
Main.getInstance(true).mediator().
setStrategy(SimpleJavaCardDLOptions.NOTHING.name());
```

The class *Main* is defined in the gui folder. After defining the parameters, we can launch the heuristics. We define also an listener, in order to be able to capture the event, when the heuristics terminate.

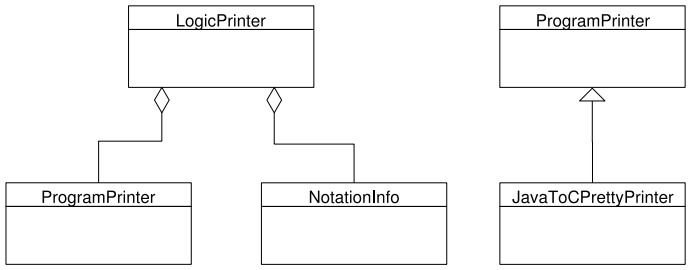


Figure 36: The attributes of the logic printer, and the inheritance of the ProgramPrinter.

```
private class NotificationListener
implements AutoModeListener
{
   public void autoModeStarted(ProofEvent e)
   { .. }
   public void autoModeStopped(ProofEvent e)
   { .. }
}
NotificationListener aListener = new NotificationListener();
Main.getInstance(true).mediator()
.addAutoModeListener(aListener);
Main.getInstance(true).mediator().startAutoMode();
```

If the heuristics execution terminates, we apply the simplifier to eventually close some more goals.

```
Main.getInstance(true).applySimplificationOnGoals();
```

### 4.2.4 Extending the LogicPrinter

An important task of our plugin is the transformation of the problem in KeY to a BLAST problem. The conceptual approach is outlined in figure 30. Here, we explain some technical aspects of the problem transformation.

The tools to print objects of type *Term* in the KeY framework are encapsulated in the *LogicPrinter* class. In order to represent terms in C notation, we create our own *LogicPrinter*.

First, we have to translate JAVA program into a C program. We extend the *ProgramPrinter*, specialized on JAVA source code, by our own *JavaToCPrettyPrinter*. For every JAVA code construct that we can't translate, an exception is thrown in the *JavaToCPrettyPrinter*. The user is informed of the problem by the interface.

Second, we also translate terms in use as preconditions and postconditions. A term like $a = 5\&b >= 3$ in KeY syntax should be translated into $a ==$

5&&$b >= 3$, in order to respect C syntax. The simplest way here is to use the *NotationInfo* object, given by the framework. It provides methods to define the syntax. We present here a sample of our definitions.

```
NotationInfo aNI = NotationInfo.createInstance();
aNI.createPrefixNotation(Op.NOT, "!");
aNI.createInfixNotation(Op.AND, "&&");
aNI.createInfixNotation((Function) aNS.lookup
(new Name("leq")), "<=");
```

The *NotationInfo* and the *JavaToCPrettyPrinter* object can be given to the *LogicPrinter* by the constructor.

# 5 Future Work

## 5.1 Programming Language Related

### 5.1.1 Switch Case Statements

*switch case* is not supported in our plugin so far. The problem is its influence on the control flow of a program. The class *CProgramCodeWizard* cannot keep track of the program locations correctly, if this statement is in use. It is possible to extend the class for that purpose, but we gave more attention to the *if then else* construct. A direct translation of a *switch case* statement into an *if then else* statement is not a good solution, because the *break* keyword provides additional freedom to the *switch case* statement.

### 5.1.2 Reconstruction of JAVA Features in C

Some features of the JAVA language cannot be translated directly to C. An example is the exception handling. However, it is possible to capture the impact of such statements on theorem proving. The statement in question could then be translated into C, such that the logic effect on the proof remains the same.

## 5.2 BLAST Related

### 5.2.1 BLAST Tuning

BLAST is a sophisticated and complex tool. It contains source-code and byte-code of other theorem provers, such as *Foci*. The mode of operation can be influenced by many parameters. We use in our plugin the option *-fmc*, to indicate that we want to use the *Foci model checker*. Further, we use the options *-craig 1 -scope* for the Craig interpolation. However, a better understanding of the BLAST tool and its possibilities and limits is desirable, because this might enhance the power of our plugin.

### 5.2.2 Invariant Optimization

Invariant optimization can accelerate the time of execution of the heuristics in KeY. This gets important for bigger problems. The only optimization of our invariant algorithm is the fact that we ignore loop-condition states being leafs. We know that their annotations can already be found on internal nodes. We believe that for complex loop-conditions, other optimization exist. Another optimization potential lies in the annotations of BLAST. Logically, they are always correct. However, it happens that the annotation encodes a fact in a complicated way. For example, $x \geq 0 \wedge x \leq 0$ can be written directly as $x = 0$.

### 5.2.3 Error Traces

If BLAST finds a feasible path to an error location in the ART, it generates a so called counter example. The counter example represents an execution of the

program, leading to an error. This information could help the user to redesign his program or the specification. Unfortunately, the counter examples are hard to understand. However, the effort to translate them into a human-readable format would be a great asset for the KeY system.

## 5.3 User Interface Related

### 5.3.1 Style

The look of the user interface was not a priority of this project. In order not to confuse the users, it should be adapted to the general style of KeY. The window management is not very good at the moment. A better strategy here would improve the experience of the user.

### 5.3.2 Input

We don't control the input directly. We wait until the user tries to apply the values. We rely on the fact, that the application of a taclet fails and throws an exception, if bad values have been specified. The taclet parameter input window of KeY uses an approach, that is more user friendly. It controls the values when the user is writing. This is a clear advantage, because eventual errors show up immediately. It would be good to provide the same service in the plugin.

## 5.4 Theory Extensions

### 5.4.1 Variant Discovery

The user has to specify the variant himself so far. We could argue, that this is not a real problem because it's not too difficult to find the variant of a problem. This is certainly true for simple problems, but the variant gets easily complicated. The importance of variant discovery is also given, because the final goal is to achieve a full automatization. The common user doesn't want to learn theories, he just wants to solve his problem as fast as possible.

### 5.4.2 Multiple Loops

Our plugin supports only simple loops so far. However, it is possible to extract invariants for nested loops and loop sequences from an ART. A good way to overcome this limitation might be a analysis of the problem structure. The analysis should provide information on how the loops are nested and cascaded. Given that information, the invariants could be searched first. Afterwards, they might also guide the plugin on how to apply taclets to resolve the problem.

### 5.4.3 Loops in Context

We assume, that the loop is isolated and not in a context of other statements. An enhancement of the plugin would be to allow other statements before and after the loop.

# 6 Conclusion

Two basic verification techniques are theorem proving and model checking. Theorem proving is powerful, but difficult to use. Model checking is fully automatic, but less powerful and hard to extend. We found a possibility to combine advantages of both approaches.

We propose a method to increase the degree of automation in loop proving. Using the model checker BLAST, we can find loop invariants for problems of a given form. This allows the user, to show partial correctness automatically in the theorem prover KeY. In contrast to model checking, the user can go further and show complete correctness, by specifying the variant.

In order to support our method, we identify the equivalence between a path formula in model checking and symbolic execution of source code in theorem proving. Further, we present an important property of state annotations in BLAST reachability trees. These theoretical basics allow us to explain, why the invariant discovery method we propose is correct.

We implemented the invariant discovery algorithm as a plugin for the KeY system. We focus on single loop problems. Because KeY is a tool for JAVA and BLAST is a tool for C, we had to convert the proof goals. This led to some restrictions on what JAVA statements the plugin is able to treat.

We applied the algorithm successfully on different sample problems. At the current state, we don't find the best invariant in every case. Nevertheless, if the problem is complicated, automated invariant discovery is a real asset, especially for unexperienced users.

# 7 Appendix, Example Database

We present in this chapter a selection of six problem examples. The examples are selected in a way, to illustrate different properties a loop problem can have. We present for each example a description and the ART.

## 7.1 Single Decrease

| | |
|---|---|
| Description | $z1$ is initially greater than $n$. A while loop decrements it down to $n$, using steps of one. The postcondition demands $z1$ to be $n$, after the loop execution. The ART of this problem is in figure 37. |
| Classification | - simple loop<br>- one variable modified |
| KeY Proof Goal | `n_pre_0 < z1_pre_0`<br>`==>`<br>`{n:=n_pre_0, z1:=z1_pre_0}`<br>`\<{ while ( z1>n ) { z1--; } }\> z1 = n` |
| Optimal Invariant | $z1 \geq n$ |
| Invariant Found | $z1 > n \vee (n = z1 \vee (z1 > n \wedge n \neq z1)))$ |
| Degree of Automation | Variant $z1 - n$ has to be specified. |

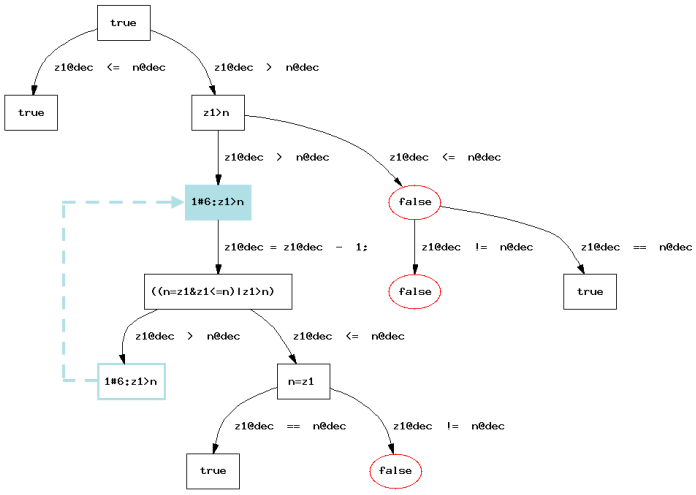


Figure 37: ART of a program decreasing a variable to n.

## 7.2 Triple Increase

| | |
|---|---|
| Description | $z1$ is initially smaller than $n$. A while loop increments it up to $n$, using steps of three. The postcondition demands $z1$ to be bigger or equal to $n$ and smaller than $n+3$. The BLAST ART is given in figure 38. |
| Classification | - simple loop<br>- one variable modified<br>- increment step is three |
| KeY Proof Goal | `z0_0 < n0_0`<br>`==>`<br>`{n:=n0_0, z1:=z0_0}`<br>`\<{ while ( z1<n ) { z1=z1+3; } }\>`<br>`(!z1 < n & z1 < n + 3)` |
| Optimal Invariant | $z1 \leq n+2$ |
| Invariant Found | $z1 \leq n+2 \vee z \leq n-1$ |
| Degree of Automation | Variant $n - z1$ has to be specified. |

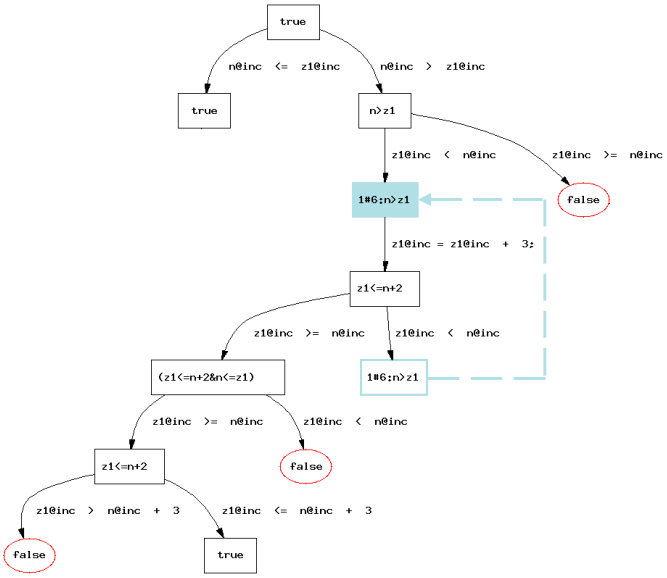


Figure 38: ART of a program increasing a variable to n by steps of three.

## 7.3 Addition

| | |
|---|---|
| Description | $z2$ is added to $z1$ in this program. $z2$ is decremented to 0 in the loop. Meanwhile, $z1$ is incremented. Finally, $z1$ is equal to the sum, we calculate in the beginning. The corresponding ART is given in figure 39. |
| Classification | - simple loop<br>- two variables modified |
| KeY Proof Goal | `z1_pre_0 >= 0, z2_pre_0 >= 0,`<br>`==>`<br>`{res:=z1_pre_0+z2_pre_0, z1:=z1_pre_0,`<br>`z2:=z2_pre_0}`<br>`\<{ while ( z2>0 ) { z2--; z1++; } }\>`<br>`z1 = res` |
| Optimal Invariant | $res = z1 + z2 \wedge z2 \geq 0$ |
| Invariant Found | $((res = z2 + z1 \wedge res = z1 \wedge z2 \leq 0) \vee (res = z2 + z1 \wedge z2 > 0)) \vee ((res = z2 + z1 \wedge res = z1) \vee (1 \leq z2 \wedge res = z2 + z1 \wedge res \neq z1)))$ |
| Degree of Automation | Variant $z2$ has to be specified. |

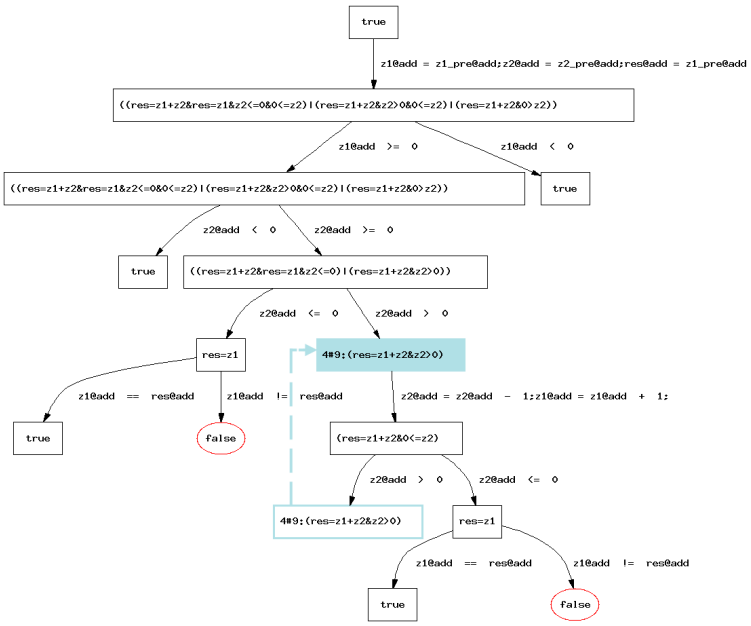


Figure 39: ART of a program adding two variables.

## 7.4 Nested Loop

| | |
|---|---|
| Description | Two variables $z1$ and $z2$ are decremented. $z1$ is decremented in the outer loop. For each iteration in the outer loop, $z2$ is decremented from $z1$ to zero in the inner loop. The ART is given in figure 40. |
| Classification | - nested loop<br>- two variables modified |
| KeY Proof Goal | `z2_pre_0 = 0, z1_pre_0 >= 0`<br>`==>`<br>`{z1:=z1_pre_0, z2:=0}`<br>`\<{`<br>`  while ( z1>0 )`<br>`  {`<br>`  z1--;`<br>`  z2=z1;`<br>`  while ( z2>0 ) { z2--; }`<br>`  }`<br>`}\> (z1 = 0 & z2 = 0)` |
| Optimal Invariant | $z1 \geq 0 \wedge z2 = 0$ |
| Invariant Found | $(z1 \geq 0 \wedge z2 = 0) \vee ((z2 = 0 \wedge z1 = 0 \wedge z1 \leq 0) \vee z1 > 0)$ |
| Degree of Automation | Variant $z1$ has to be specified. Inner loop does block the automatic proof of two sub-goals. |

## 7.5 Decimal Number Simulator

| | |
|---|---|
| Description | Two variables $z1$ and $z2$ are decremented, such they behave like a decimal number together. Every-time if $z2$, the second digit of the number, is zero, we set it to 9 and decrease $z1$. The ART is given in figure 41. |
| Classification | - simple loop<br>- two variables modified<br>- case distinction |

| | |
|---|---|
| KeY Proof Goal | `z2_pre >= 0, z1_pre_0 >= 0`<br>`==>`<br>`{z1:=z1_pre_0, z2:=z2_pre}`<br>`\<{`<br>`  while ( z1>0\|\|z2>0 )`<br>`  {`<br>`  if (z2==0) { z1--; z2=9;}`<br>`  else { z2--; }`<br>`  }`<br>`}\> (z1 = 0 & z2 = 0)` |
| Optimal Invariant | $z1 \geq 0 \wedge z2 \geq 0$ |
| Invariant Found | $(0 \leq z2 \wedge 0 \leq z1) \vee (0 \leq z2 \wedge 0 \leq z1 \wedge z1 \leq 0)$ |
| Degree of Automation | Variant $z1 \cdot 10 + z2$ has to be specified. |

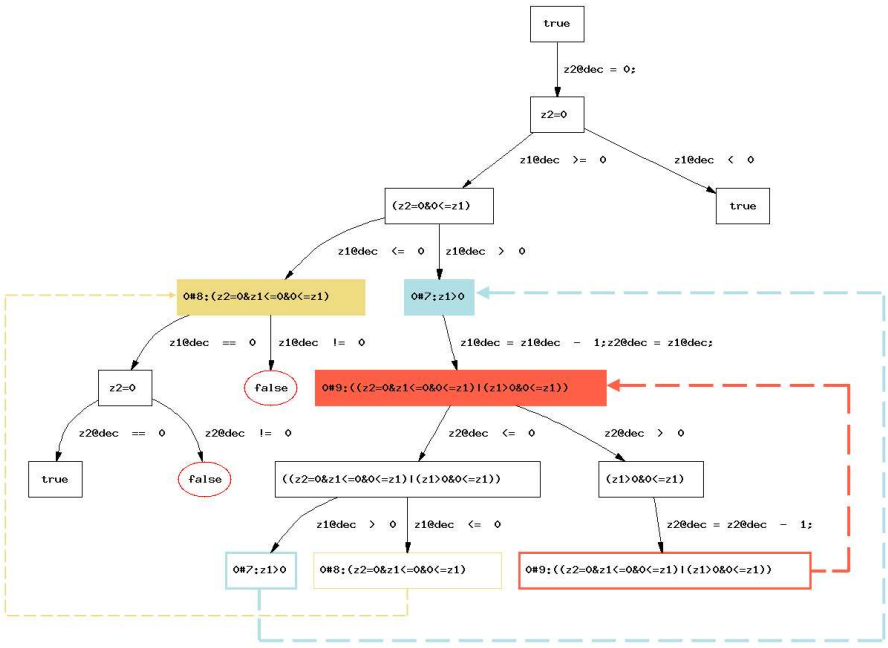


Figure 40: ART of a program containing a nested loop.

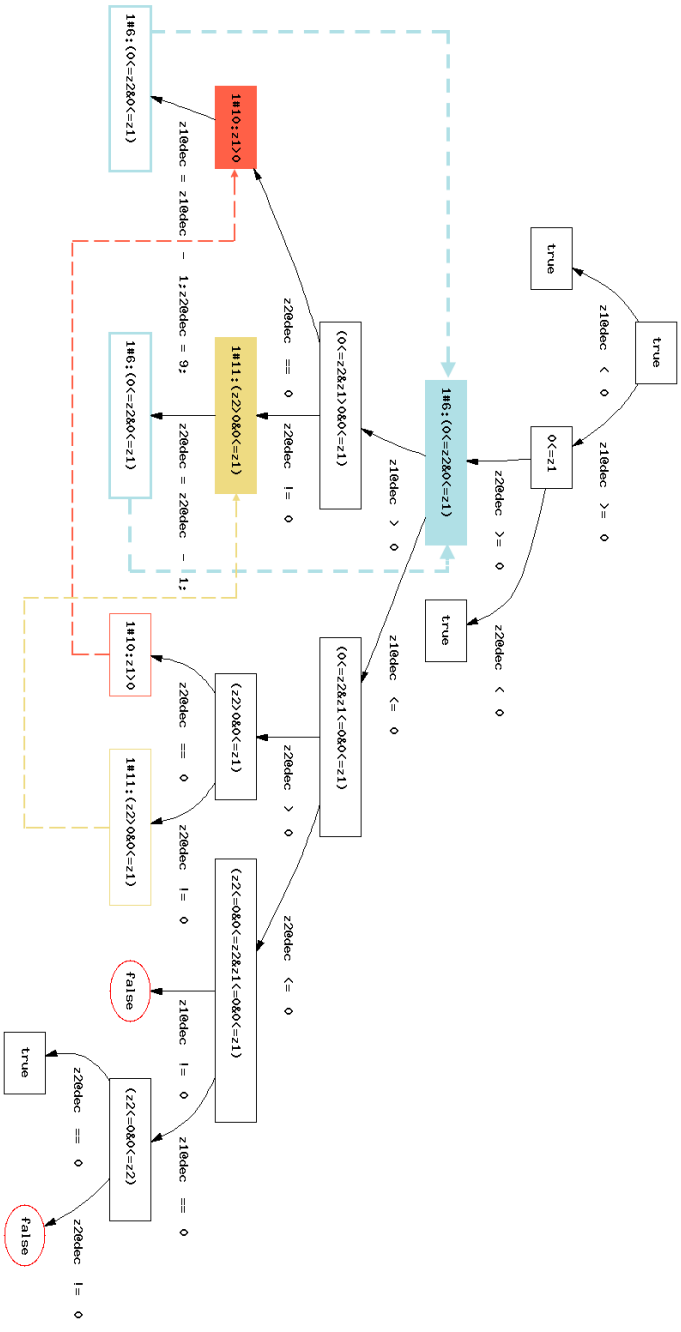


Figure 41: ART of a program simulating a decimal number.

## 7.6 Complex Decrease

| | |
|---|---|
| Description | Two variables $z1$ and $z2$ are decremented. $z2$ is the control variable. It triggers the behavior of $z1$, the variable tested in the postcondition. The ART for this example is given in figure 42. |
| Classification | - simple loop<br>- two variables modified<br>- triple case distinction<br>- complex postcondition |
| KeY Proof Goal | `0 < n_pre_0, z1_pre_0 >= 0, z2_pre_0 => 0,`<br>`m_pre_0 >= 0, n_pre_0 >= m_pre_0`<br>`==>`<br>`{m:=m_pre_0, n:=n_pre_0,`<br>`z1:=z1_pre_0, z2:=z2_pre_0}`<br>`\<{`<br>`  while ( z1>0 )`<br>`  {`<br>`  if (z2>0){ z2=z2-1; z1=z1-1; }`<br>`  else if (z2==m){ z1=-40; }`<br>`  else { z2=n; z1=z1-5; }`<br>`  }`<br>`}\> (!0 < z1 & -5 < z1 \| z1 = -40)` |
| Optimal Invariant | $z1 = -40 \vee -4 \leq z1$ |
| Invariant Found | $0 \leq z1 \vee z1 = -40 \vee -4 \leq z1$ |
| Degree of Automation | Variant $z1$ has to be specified. |

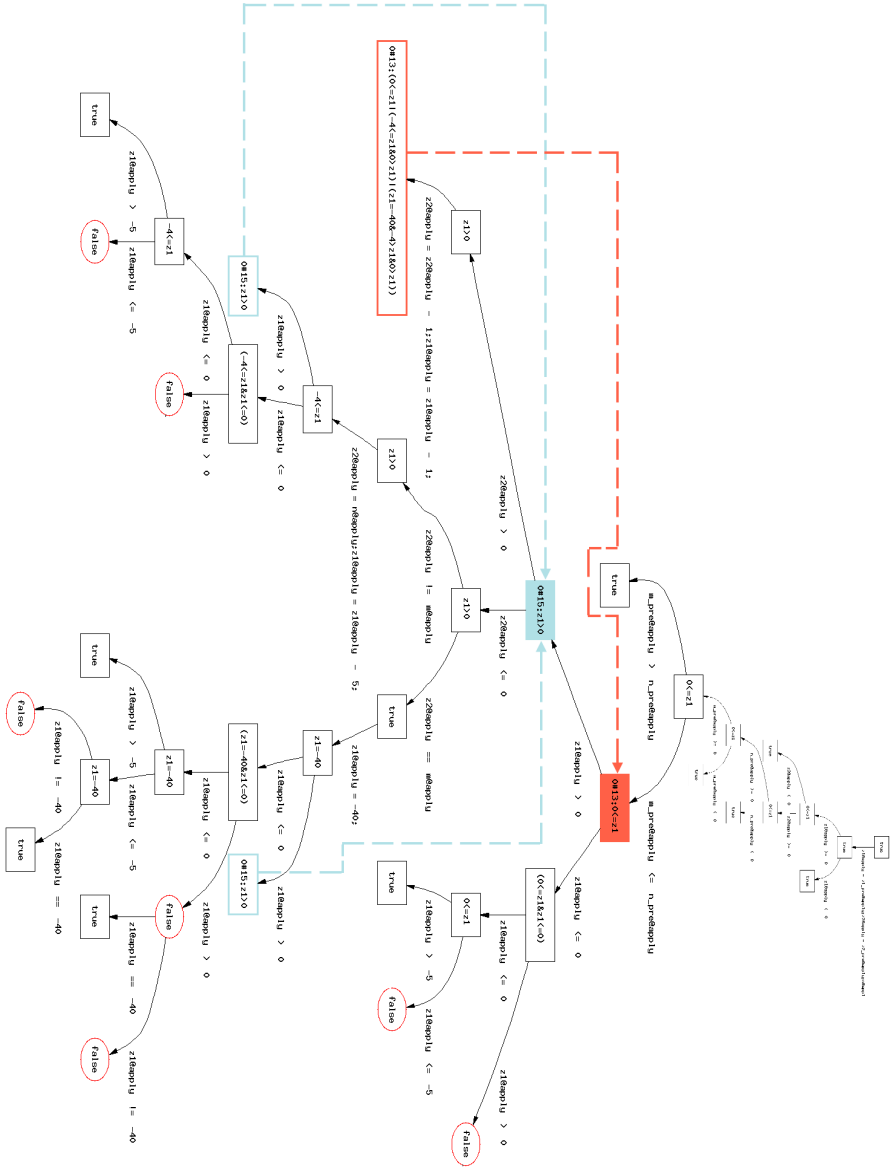

Figure 42: ART of a program having a triple case distinction in the loop body.

# 8 Acknowledgments

This thesis would not be possible without the two excellent projects on KeY and BLAST. Particularly, we would like to mention here Richard Bubel and Steffen Schlager of the KeY team, Dirk Beyer and Thomas Henzinger of the BLAST team. The most important contribution however is due to the project supervisor, Thomas Baar. We appreciate his effort very much.

## References


[1] Hasan Amjad. *Combining model checking and theorem proving.* PhD thesis, University of Cambridge, 2004.

[2] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating model checking and theorem proving for relational reasoning. In *Seventh International Seminar on Relational Methods in Computer Science (RelMiCS 2003)*, volume 3015 of *Lecture Notes in Computer Science (LNCS)*, pages 21–33, Malente, Germany, May 2003 2003.

[3] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[4] Bernhard Beckert, Steffen Schlager, and Peter H. Schmitt. An improved rule for while loops in deductive program verification. In Kung-Kiu Lau, editor, *Proceedings, Seventh International Conference on Formal Engineering Methods (ICFEM), Manchester, UK*, LNCS 3785, pages 315–329. Springer, 2005.

[5] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with Blast. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005, Edinburgh, April 2-10)*, LNCS 3442, pages 2–18. Springer-Verlag, Berlin, 2005.

[6] Zhiqun Chen. *JAVA CARD Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley, 2000.

[7] W. Craig. Linear reasoning: A new form of the herbrand-gentzen theorem. *Journal of Symbolic Logic*, 22:250–268, 1957.

[8] Formal methods virtual library, http://vl.fmnet.info/.

[9] Graphviz - graph visualization software, http://www.graphviz.org/.

[10] Reiner Hähnle, Peter H. Schmitt, and Bernhard Beckert. *The Key Book – The Road to Verified Software.* Springer, 2006. To appear.

[11] Adaptive Ltd., Boldsoft, IBM, Iona Technologies, and Object Management Group. *UML 2.0 OCL Specification*, 2003.

[12] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[13] Jacques Zahnd. *Logique élémentaire.* Presses polytechniques et universitaires romandes, 1998.

# List of Figures