Karlsruhe Institute of Technology

Trace Based Reasoning with KeY and JML

Studienarbeit
von

Andreas Wagner

An der Fakultät für Informatik
Institut für Theoretische Informatik (ITI)

Verantwortlicher Betreuer:	Prof. Dr. ren. nat. B. Beckert
Betreuer:	Daniel Bruns

17. Juni 2013

Abstract

Program specification and verification based on the concept of Hoare triples [Hoa69] defines the relation of the output to the input and treats the program as black box. This thesis deals with a type of specification that allows to consider intermediate steps of the program execution. The logic of *Dynamic Trace Logic* [BB13] is extended to JavaDTL to support Java Card as well as important Java features and the Java Modeling Language is extended to support defining properties in this language. A comparison with other tools is made that also deal with temporal behavior ofprograms. Examples for an implementation that is based on this logic together with sequent calculus are presented.

Contents

1 Introduction

Topic of this minor thesis is the question how verification of temporal properties of Java programs can be performed. Temporal properties allow to specify in which way evaluation of a given expression changes during the course of the program run. Many software verification tools such as KeY are based on the concept of pre- and post-condition. This allows to specify the functional behavior of sequential programs. So far the approach for dealing with temporal specifications was either runtime checking [HL10] or building a simplified finite state model of the program [Hol03]. An open question is how to combine symbolical execution of programs with specification in temporal logic.

The theoretical foundation for this minor thesis is the paper *Dynamic Trace Logic* [BB13] in which an extension of the logic for KeY has been considered. Temporal operators like in linear temporal logic (LTL) have been added to KeY. In that paper a very simple, so called *while language* is used. This language is deterministic, so the the trace of the program is linear and consists of a possibly infinite number of states. The logic defined there is called dynamic trace logic(DTL).

In this thesis the concept of dynamic trace logic is used on the more complex Java language. The focus hereby is the evaluation of a prototypical implementation that is used to show how to verify Java programs using trace semantics. Several examples were created by the author (see section 9) will illustrate how this affects the proving process. These examples serve two purposes. They were used to test and debug the Java implementation and the new rules that are defined in *taclets* 6.

An new contribution to the reasearch is also an extension to the Java Modeling Language (JML) that has been invented by the author of this thesis. It enables the definition of specification containing trace semantics within the JML comments within the Java source files is described in section 7.3. This extension will also comprise a way to specify trace semantic specific loop invariants for automatic proving in the JML comments. Examples will give a demonstration of the use of this extension and illustrate how temporal properties about a program as well as loop invariants can be defined.

Instead of just looking at the input / output behavior the focus of this publication are the changes happen to some or all of the object fields during the program run. This has several reasons. For a program that does not terminate, like a server process that is always running, we could otherwise not prove anything other than the non-termination. The advantage of trace semantics is that properties about the program can be proven independently of program termination. For concurrent processes that access the same data structure at a time you can ensure consistency throughout the program run by a specification that uses temporal operators. Also for specifying information flow properties this approach is beneficial. You can specify different levels of confidence and ensure that you cannot gain any information from the output of a lower about a higher level. KeY with trace semantics combines the possibility to use temporal specification with the support for unlimited data structures such as linked lists or trees.

2 Logic

Logics are about formulating propositions and formulae, evaluating their truth values and inferring other formulae from them. Some logics are especially suited to describe and verify computer programs. This chapter gives an overview over the most important logics used for specification and verification tasks an how they relate to each other. This in mind the author introduces a new logic called *Java Card Dynamic Temporal Logic* (JavaDTL) which is the theoretical foundation for an extension of the KeY-verifier, the topic of this thesis. Basically JavaDTL is DTL adapted to the Java programming language.

In propositional logic every formula can be evaluated to either *false* or *true*. Logical atoms are combined to propositional formulae via the well-known boolean operators such as $\wedge$ (and), $\vee$ (or), $\neg$ (not), ... Most other more advanced logics are based on this one which has roots back to the ancient greek philosophers.

2.0.1 Syntax and Semantics of First Order Logic

When you add existential and forall quantifiers to propositional logic you get to first-order logic. There are the following logical symbols, i.e. symbols that always have the same meaning and are called rigid: the binary predicates $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence), the unary function $\neg$, brackets, logical variables, and the equality on terms ($=$). They can be combined to formulae which evaluate to either *false* or *true*. The existential $\exists$ and the forall quantifier $\forall$ may range over constant symbols but not over predicates. Quantification over predicates is only possible in higher order logic.

In addition to formulae there are so called terms that evaluate to a specific object of a domain. The domain may be divided into sets of different sorts; in this case the logic is called typed first-order logic. This feature can be used to model program variables of a primitive type, such as *int* or *boolean*, as well as instances of classes in object oriented programming.

Predicate symbols are used to model an n-ary relation from the domain to the logical values *true* or *false*. In case of a typed logic the sorts must match the signature in order to be well typed. 0-ary Predicate symbols are equivalent to propositional variables.

Function symbols are n-ary relations within the domain. They may be used to model the arithmetic operators as well as functions in a programming language. 0-ary functions are called constant symbols that represent an element of the domain.

2.0.2 More Advanced Logics

Propositional Logic can be extended with modalities such as *possibly* or *necessarily*. Propositions such as *Possibly it rains* and *A triangle necessarily has three corners* can be expressed. Modal operators can be added to both propositional logic and predicate logic. The modal operator for *possibly* is represented by the symbol $\Diamond$, the one for *necessarily* with $\Box$.

Dynamic logic extends modal logic with the modal operators $[\pi]$ and $\langle\pi\rangle$ where π is a program. The expression $[\pi]p$ means that p necessarily has to hold; in the case $\langle\pi\rangle p$ the proposition possibly holds. There is an infinite number of

possible modalities so dynamic logic is a case of multimodal logic. The two forms of modalities can be converted in each other by the use of the negation operator as shown in the following equivalences: $[\pi]p \leftrightarrow \neg\langle\pi\rangle\neg p$ and $\langle\pi\rangle p \leftrightarrow \neg p[\pi]\neg p$.

Java Dynamic Logic (JavaDL) is the Logic KeY operates on. It is uses the program modalities $[\pi]$ and $\langle\pi\rangle$ from dynamic logic however with a different semantic. The proposition following $[\pi]$ has to hold only in cases where π terminates for the formula to evaluate to true. The formula $\langle\pi\rangle p$ is true in cases where the program π terminates and p holds after program termination.

Java Card Dynamic Logic has also been enriched with special features of Java to handle assignments, objects, arrays, etc. As for now only deterministic Java programs are considered. A possible source of indeterminism in Java is the use of threads where different interleaving by the scheduler can lead to different results. JavaDL has first been described in *A Dynamic Logic for the Formal Verification of Java Card Programs* [Bec01].

2.1 Difference Between Determinism and Indeterminism

An algorithm is called deterministic if for the same input always the same computational steps are performed. In the problems considered for verification in this thesis the complete input is given in the beginning. Such algorithms are called *offline algorithms*. For deterministic programs the next computational step is only dependent of the program *state*, i.e. the values of the entity of program variables.

The execution tree that describes all possible program execution paths for a fixed input is in the case of an deterministic program just a linear list, the so called *trace*. In case the program terminates the trace is finite; for nonterminating programs it is infinite. The terms *state* and *trace* are explained more in detail in section 6 and 9 .

A program is called indeterministic if it isn't completely deterministic. Possible sources of indeterminism are interleaving of parallel code and the use of (pseudo-)random variables for randomized algorithms. Also indeterminism can occur together with the use of float variables because operations such as multiplication and addition are not exactly commutative and the compiler may do such transformations. Indeterministic programs are not considered in this thesis but are a possible area for further work on the KeY-verifier.

3 Dynamic Trace Logic for Java (JavaDTL)

Dynamic Trace Logic is a logic that combines temporal logic with dynamic logic. It uses the modal operators $\Box$ (alway), $\Diamond$ (eventually), $\bullet$ (weak next), $\circ$ (strong next), **U** (until), **W** (weak until) and **R** (release) . The semantic of these symbols are closely related to LTL. It also uses a program modality [[Prg]]. Dynamic trace logic has been described in *Dynamic Trace Logic* [BB13] for a very reduced while-language. In this section The concept of Dynamic trace logic is described. It can be adapted to Java so that it can be the logic for the KeY-program similarly to JavaDL.

3.1 Syntax of JavaDTL

LVar is the set of local program variables and *GVar* is the set of global program variables. These sets are disjoint. Object fields are considered to be global program variables whereas temporal variables declared within methods belong to the set of local program variables. Logical variables are variables that are rigid, i.e. their value is fixed throughout the program trace. Quantifiers are restricted to range over local variables. The logic of *Dynamic Trace Logic* [BB13] is restricted to a simple while language and therefor the sets of function and predicate symbols are small, fixed sets. JavaDTL allows a larger set of function and predicate symbols. Amongst them are the "Predefined Opererators in Java Card DL" found in appendix A of *"The KeY Approach"* [BHS07]. The set of function symbols includes the standard arithmetic operators, the zero-ary function symbols TRUE and FALSE and several Java-specific ones. Some of them depend on the chosen integer semantics because KeY supports different modes of handling integers in Java. Side effect free Java methods are also function symbols. Important predicate symbols are the equality predicate $\doteq$, the symbol for inequality $!\doteq$, the ordering relations and the existential predicate $\exists A$ (for an A of some type $\mathfrak{T}$)

Definition 1 (Expressions)**.** *Expressions in JavaDTL can be of any type that is allowed in Java. That can be a primitive type such as* int*,* char*,* bool*, . . . , or an object type. Expressions can be constructed by using methods and the predefined infix operators of Java. An expression is correctly constructed if a Java compile time checker would accept it. A method is called* side effect free *if its evaluation does not change any object field. Expressions that only use side effect free methods are called* side effect free*.*

Syntactically correct Java programs are the programs that are allowed to be used in JavaDTL. Only some newer and more advanced features such as generics and dynamic class loading are not supported yet. Programs can contain object orientated features and expressions may have side effects. Every assignment to a field variable of some object, i.e a global variable, is considered a state transition.

Definition 2 (Statements, programs)**.** Programs *and* statements *are defined similarly to* Dynamic Trace Logic [BB13]*, but adapted to the Java language. Statements have the form:*

- `x = a;` *where* `x` $\in$ *LVar and* `a` *is a program expression of some type* $\mathfrak{T}$ *(assignment to local variable)*
- `X = a;` *where* `X` $\in$ *GVar and* `a` *is a program expression of some type* $\mathfrak{T}$ *(assignment to global variable)*
- `if (a) {`π_1`} else {`π_2`}` *where* `a` *is a program expression of type* `bool` *and* π_1 *and* π_2 *are Java programs. The case in which the* `else` *branch is missing* (`if (a) {`π`}`) *can be seen as a special case where* π_2 *is empty.* (conditional)*, or*
- `while (a) {`π`}` *where* `a` *is a program expression of type* `bool` *and* π *is a Java program that contains at least one assignment to a global variable.*

- *Other program constructs such as* `for`*-loops or* `do {`π`} while (a)` *constructs can be rewritten equivalently with the help of* `while (a) {`π`}`. (loop)

Programs are finite sequences of statements. The empty program is called ϵ.

To represent the progress during program execution a program trace is used. Starting from an initial state the program symbolically executes program statements an passes through the states of the trace. When a statement is symbolically executed it is removed from the program π within the program modality $[[\pi]]$ and the deleted information about the program is converted into a state update to delay the substitution until the program in the program modality has completely disappeared. The difference between JavaDTL and DTL state updates is that JavaDTL allows more different types for program variables.

Definition 3 (State Transition)**.** *Formally a state transition is a relation between two states. A state transition is caused by an assignment to a global variable. In the case of deterministic programs that we consider, it is a partial function that yields for every state, except the last one, the succeeding one. Initializing the values of the method parameters for the proof obligation does not lead to a state transition. The first state of the trace contains the values of the field variables before the execution of the first line of code in the method. The last state contains the values directly after termination.*

Definition 4 (State updates)**.** *Let* x $\in$ *LVar* $\cup$ *GVar be a program variable and* a *be an expression. Then,* {x:=a} *is an update.*

{x := 0}, {x := x + 1}, {b := true}, {b := !b} and {obj := null} are some examples of updates for variables of different kind. They can be applied to JavaDTL formulae replacing the free occurrences of the specific variable.

JavaDTL formulae have the form $\mathcal{U}[[\pi]]\varphi$ where $\mathcal{U}$ is a sequence of updates, π is a Java program and φ is another JavaDTL Formula. $\mathcal{U}[[\pi]]\varphi$ expresses that φ holds over all traces τ in which the first state fulfills the conditions stated in the Updates $\mathcal{U}$. The traces over which φ has to hold are generated by symbolical execution of π. There might be several traces to consider because it may be the case that the updates $\mathcal{U}$ only partially describe the initial state.

Definition 5 (Formulae)**.** State formulae *and* trace formulae *are inductively defined in the following way:*

0. *All state formula are also trace formula.*
1. *All boolean side effect free expressions (Def. 1) are state formulae.*
2. *If* Φ *and* Ψ *are (state or trace) formulae, then the following are trace formulae:* $\Box\Phi$ *(always),* $\bullet\,\Phi$ *(weak next),* $\Phi\mathbf{U}\Psi$ *(until).*
3. *If* $\mathcal{U}$ *is an update and* Φ *a state formula, then* $\mathcal{U}\Phi$ *is a state formula.*
4. *If* π *is a program and* Φ *a trace formula, then* $[[\pi]]\Phi$ *is a state formula.*
5. *The sets of state and trace formulae are close under the logical operators* $\neg$, $\wedge$, $\forall$.

In addition the following abbreviations are used:

$$
\begin{aligned}
\Diamond\Psi &:= \neg\Box\neg\Phi \text{ (eventually)},\\
\circ\Psi &:= \neg\bullet\neg\Psi \text{ (strong next)},\\
\Phi\mathbf{W}\Psi &:= \Phi\mathbf{U}\Psi \vee \Box\Phi \text{ (weak until)},\\
\Phi\mathbf{R}\Psi &:= \neg(\neg\Phi\mathbf{U}\neg\Psi) \text{ (release)},\\
\Phi \vee \Psi &:= \neg(\neg\Phi \wedge \Psi),\\
\Phi \rightarrow \Psi &:= \neg\Phi \vee \Psi,\\
\exists x.\Phi &:= \neg\forall x.\neg\Phi.
\end{aligned}
$$

A JavaDTL formula with no program modality $[[\pi]]$ *and without temporal operators is called* non-temporal.

3.2 Semantics of JavaDTL

A *state* is a construct that contains all the information about a program in execution at a specific time, whereas the *program trace* contains the information about the ordering of the states. Expressions and JavaDTL formula are evaluated over program traces and variable assignments. Program traces determine the value of global and local program variables of the set *GVar* and *LVar* whereas *variable assignments* set the value of logical variables.

Definition 6 (States, variable assignments)**.** *A* state s *is a function assigning a value of the correct type to every location* loc *and every local variable* ll*, formally:* $LVar \cup GVar \rightarrow \mathcal{T}$*, where* $\mathcal{T}$ *is a legal Java type.* $\mathcal{T}$ *can be either a primitve or an object type. A location is a field of an object that exists on the Java heap.*

A variable assignment β *assigns values to logical values of the set* V*, i.e. :* $\beta : V \rightarrow \mathcal{T}$*.*

The notation $s\{x \mapsto d\}$ is used to express a state that is like s but with the value of variable x changed to $d \in \mathcal{T}$. By $\beta\{x \mapsto d\}$ and $\tau\{x \mapsto d\}$ changes to variable assingments and traces are expressed. Locations can be of type *object*, any subtype thereof or of a primitive type like *int* or *bool*. A local variable can have the same types as a location but is stored on the *stack* instead. Variables declared within a method as well as the *function parameters* are local variables. Linear temporal formulae can only speak about the values of locations but not about local variables. The values of local variables are considered to be hidden from the logic. However within the program modality they can occur both on the left and the right hand side of assignments. Assignments are handled by special assignment rules which symbolically execute them.

Definition 7 (Traces)**.** *A program trace is a sequence of states. It represents a single run of a program. The program execution is considered to be a linear trace between program states. This trace may be finite or infinite. The traces do not branch because only deterministic programs are considered. Infinite traces correspond to non-termination. figure 3.3 shows a visualization of traces in which states are represented by circles and state transitions by arrows.*

Definition 8 (Semantics of expressions)**.** *For a* state s *and a variable assignment* β*, the valuation* $e^{s,\beta}$ *of an expression* e *in a state* s *is of a Java type*

$\mathcal{T}$ and is determined by interpreting program variables v *in the state* s. *logical variables are assigned values to by $\beta(x)$. For the function and relations the interpretation is used that is specified in the Java standard. For integer operators the interpretation depends on how the settings within KeY for handling integers is. KeY allows handling integers as mathematical integers as well as two different modes that account for the finite range of Java integers and overflows.*

Program expression without logical variables do not depend on β and hence e^s is written instead of $e^{s,\beta}$. In case e *is an expression of type bool, $s, \beta \models e$ is written to express that $e^{s,\beta}$ is true. $s \models e$ denotes that e^s is true in the case of β-independent logical variables .*

Assignments to global variables leads to new observable states. The observables states are the states within the program trace τ. To prove a formula $[[\pi]]\Phi$ valid it has to be shown that Φ holds for the trace generated by the program π. If Φ contains temporal operators it is evaluated over the whole trace whereas in the case without temporal operators it is evaluated in the first state of the trace. In every program state each object field has a determined value. Assignments to field variables causes a state transition. The formal definition of a trace as described in *Dynamic Trace Logic* [BB13] is shown in the follwing definition:

Definition 9 (Program Trace)**.** *Given an (initial) state s, the trace of a program pi is defined as shown in figure 1*

Figure 1: Formal definition of a trace

$$
\begin{aligned}
trc(s, \epsilon) &= \langle s \rangle \\
trc(s, v = x; \omega) &= trc(s\{v \mapsto x^s\}, \omega) \\
trc(s, G = x; \omega) &= \langle s \rangle \cdot trc(s\{G \mapsto x^s\}, \omega) \\
trc(s, \text{if}(e)\{\pi_1\}\text{else}\{\pi_2\}\omega) &= \begin{cases} trc(s, \pi_1\omega) & if\, s \vDash e \\ trc(s, \pi_2\omega) & if\, s \nvDash e \end{cases} \\
trc(s, \text{while}(a)\{\pi\}\omega) &= \begin{cases} trc(s, \text{while}(a)\{\pi\}\omega) & if\, s \vDash e \\ trc(s, \omega) & if\, s \nvDash e \end{cases}
\end{aligned}
$$

This definition can be adapted to the trace of a Java program. A special case is the case of a loop that does not contain any assignment to a global variable. In this special case it is described in *Dynamic Trace Logic* [BB13] that a NOP assignment is performed to guarantee progress. The trace of the program is then modified in a way that it contains states with identical value for every loop iteration. These additional state transitions between states with identical values do not change the value of simple formulae like $\Box p$ or $\Diamond p$. However if you define for example the property, that some field x increases strictly in each step with the formula:
$\Box \exists int\ u;\ (u = x \wedge (\bullet u < x))$

Then this property can be broken by the insertion of the additional states. These insertions may happen due to a loop that contains just some computation on local variables. However such a temporal property is rather artificial and the problem can be solved by changing the specification to just demand $u \leq x$.

Definition 10 (Semantics of state formulae)**.** *Let s be a state and β be a variable assignment.*

$$\begin{aligned}
s,\beta \vDash e \quad & \textit{iff} \quad e^{s,\beta} = \textit{true} \\
s,\beta \vDash \neg\Phi \quad & \textit{iff} \quad s,\beta \nvDash \Phi \\
s,\beta \vDash \Phi \ \wedge \ \Psi \quad & \textit{iff} \quad s,\beta \vDash \Phi \textit{ and } s,\beta \vDash \Psi \\
s,\beta \vDash \forall x.\Phi \quad & \textit{iff} \quad \textit{for every } d \in \mathbb{Z} : s,\beta\{x \mapsto d\} \vDash \Phi \\
s,\beta \vDash [[\pi]]\Phi \quad & \textit{iff} \quad trc(s,\pi),\beta \vDash \Phi \\
s,\beta \vDash \{v := x\}\Phi \quad & \textit{iff} \quad s\{v \mapsto x^s\},\beta \mapsto \Phi
\end{aligned}$$

A state formula Φ is valid *if $s,\beta \mapsto \Phi$ for all s and all β.*

Definition 11 (Semantics of trace formulae)**.** *Let τ be a trace and β a variable assignment.*

$$\begin{aligned}
\tau,\beta \vDash \neg\Phi \quad & \textit{iff} \quad \tau,\beta \nvDash \Phi \\
\tau,\beta \vDash \Phi \ \wedge \ \Psi \quad & \textit{iff} \quad \tau,\beta \vDash \Phi \textit{ and } \tau,\beta \vDash \Psi \\
\tau,\beta \vDash \forall x.\Phi \quad & \textit{iff} \quad \textit{for every } d \in \mathbb{Z} : \tau,\beta\{x \mapsto d\} \vDash \Phi \\
\tau,\beta \vDash \Box\Phi \quad & \textit{iff} \quad \tau[i,\infty),\beta \vDash \Phi \textit{for every } i \in [0,|\tau|) \\
\tau,\beta \vDash \Phi\mathbf{U}\Psi \quad & \textit{iff} \quad \tau[0,i),\beta \vDash \Box\Phi \textit{ and } \tau[i,\infty),\beta \vDash \Psi \textit{ for some } i \in [0,|\tau|) \\
\tau,\beta \vDash \bullet\,\Phi \quad & \textit{iff} \quad \tau[1,\infty),\beta \vDash \Phi \textit{ or } |\tau| = 1 \\
\tau,\beta \vDash \gamma \quad & \textit{iff} \quad \tau[0],\beta \vDash \gamma \textit{(in case } \gamma \textit{ is a state formula)}
\end{aligned}$$

A trace formula Φ is valid *if $\tau,\beta \mapsto \Phi$ for all τ and all β.*

To express properties of these program traces, dynamic trace logic contains several operators. They are explained in the following with the help of example traces. In the tables that come with these explanations the columns represent the states of the trace. A cell is colored grey if the formula on the left evaluates to *true* in this state. Respectively white cells mark that the formula evaluates to *false*.

Box

$\Box p$ means that the property p has to hold in the entire subsequent path for this formula in order to be valid. The expressions p and q in the next examples have to be a side effect free and of type *boolean*.

Diamond

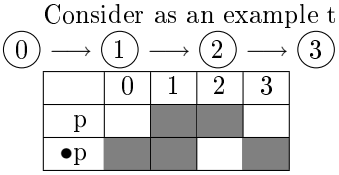
$\Diamond p$ means that the property p has to hold in at least one step in the future of the trace for this formula to be valid.

WeakNext

$\bullet p$ means that the property p has to hold in next step. If the current state is the last state then the formula is automatically valid. In the following tables gray

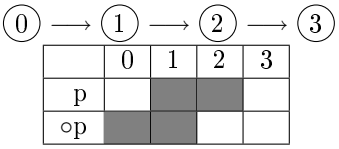
indicates that the expression in the left column is true and white corresponds to false.

Consider as an example the following finite trace:



StrongNext

$\circ p$ means that the property p has to hold in next step. If the current state is the last state then the formula is automatically invalid. Consider again the example from the previous operator. The difference here is that $\circ p$ evaluates to false in the last state.



Until

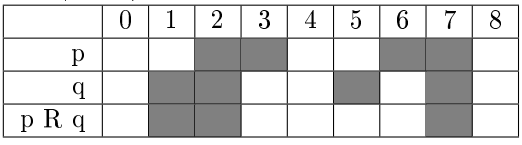
The semantic of **U** *p until q* is the following: q holds in the current state or in a future state and p has to hold until that state. From that state p does not has to hold any more. Here an example:

WeakUntil

p **W** q is similar to p **W** q except that it is also allowed that p holds for ever and q never becomes true. So the following equivalence holds:
p **W** $q \Longleftrightarrow p$ **U** $q \vee \Box\, p$

Release

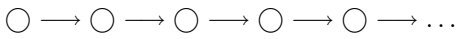
The meaning of p **R** q is: p releases q if q is true until the first position in which q is true (or forever if such a position does not exist). p **R** q is equivalent to q **W** $(p \wedge q)$. Here an example:



3.3 Using Java DTL in Sequent Calculus

One way to prove the validity of JavaDTL formula is the use of sequent calculus. In Sequent Calculus the formula is manipulated by the application of rules. These rules simplify the formulae to the point where the validity can be shown

Figure 2: structure of a trace


an infinite trace


a finite trace

```
public class Test {
  public int num;

  public void start() {
    while(true) {
      num = num;
    }
  }
}
```

Listing 1: Test.java

by axioms. Some rules split the proof up into several simpler formulae that all have to be shown to be valid for proving the validity of the initial formula. For DTL there exists a set of rules that are described in *Dynamic Trace Logic* [BB13] and are complete and sound. It is possible to use several traces in one proof sequent. A small example is shown below.

Figure 3: Proof Tree for Example with Two Traces

$$\dfrac{\dfrac{\dfrac{}{[\![\pi]\!] \bullet \Box\Diamond\phi, [\![\pi]\!]\Diamond\phi \Rightarrow [\![\pi]\!]\Diamond\phi}\,\text{close}}{[\![\pi]\!] \bullet \Box\Diamond\phi \ \wedge\ [\![\pi]\!]\Diamond\phi \Rightarrow [\![\pi]\!]\Diamond\phi}\,\text{andLeft}}{[\![\pi]\!]\Box\Diamond\phi \Rightarrow [\![\pi]\!]\Diamond\phi}\,\text{unwindBox}$$

Only assignments to object fields cause a state transition. Assignments to local variables do not change the state. As there are usually a lot of assignments to field variables in a program you might also specify some labels for certain states. So speaking about certain states e.g. all states after an iteration of a loop, is easier. To achieve this the JML has to be extended as shown in chapter 7.3 .

The example displayed in Listings 1 and 2 can be proven in only 4 steps without even needing the *method call* rule. This example is just for demonstrating that two traces in one sequent are possible. A practical use of this feature is the definition of information flow properties.

```
\javaSource "source/";

\programVariables {
  Test p;
}

\problem {
  \[[ {
    p.start();
  }
  \]] (\diamond\box(true))
  ->
  \[[ {
    p.start();
  }
  \]] (\diamond(true))
}
```

Listing 2: TwoTraces.key

4 JML

The Java Modeling Language (JML) [LPC+08] is a modeling language to specify properties of Java code. Syntactically it is similar to Java but has some special features like ∀ and ∃ quantifiers. The JML specification is typically written in the same file as the source code and lives in a special form of comments that start with the symbol @. JML annotations are ignored by the compiler but recognized by JML tools. With the KeY System it is possible to use the specification written in JML to generate proof obligations. An extension of the JML which allows to specify properties containing trace logic is presented in this paper.

4.1 Expressions

In JML preconditions, postconditions and invariants are of type `boolean`. All valid Java boolean expressions without side-effects are allowed to be used at these places. There are also additional features in JML that do not exist in plain Java. An important feature are the quantifiers ∀ and ∃. The syntax is `(\forall T x; b1; b2)`. Which means that for all `x` of type `T` the following has to be true: If `b1` holds than `b2` holds. `b1` and `b2` are boolean expression. Similarly the expression `(\exists T x; b1; b2)` means that there is an `x` of type `T` such that `b1` and `b2` hold. With the keyword `\result` you can speak about the result value in the postcondition, with `\old(e)` about the value of an expression e before the execution of a method. With `\type(e)` you can refer to the type of an expression. In contrast to classical logic, in JML boolean values are three-valued: *true*, *false* and *undefined*. An undefined value means that an exception is thrown (e.g. division by zero or array index out of bounds). In practice you try to write your specification in a way that avoids values becoming undefined.

4.2 Method Contracts

A method contract begins with the keyword `public normal_behavior` or `public exceptional_behavior`. A normal behavior contract may not throw an exception whereas the exceptional behavior case is used when an exception is expected. With the keyword `also` two or more specification cases can be linked together.

The basic behavior of a method is described by the pair of boolean expressions `requires` and `ensures`. After the keyword `requires` there has to be a boolean expression that describes which states the specification case speaks about. Iff the expression after the requires clause holds then the boolean expression after the `ensures` clause has to hold, too. If the `requires` or the `ensures` clause is omitted the default for each of them is true.

In JML the `assignable` clause describes what may change during the execution of the method. The assignable clause is a convenient way to avoid writing `var=\old(var)` for every variable `var` that has to stay the same in the JML specification. This shortens the specification and makes it less likely to forget the specification for non-changing variables.

`diverges b` means that the program may only run indefinitely (i.e diverge) if `b` is true. Usually `b` is either `true` or `false`. `diverges true` means that the program does not have to terminate. The default `diverges false` enforces program termination in any case.

4.3 Class Invariants and Other Features

A class invariant begins with the identifier `invariant` followed by a side-effect free boolean expression `b`. The expression `b` has to be true before and after each method call of the object the class invariant belongs to. A class invariant can have an identifier such as `public`. A public invariant can only speak about public fields of other objects. To overcome this problem you can mark private fields with the identifier `spec_public`. This way they are accessible for the verification but still private for the program. Class invariants can also be declared as `static`.

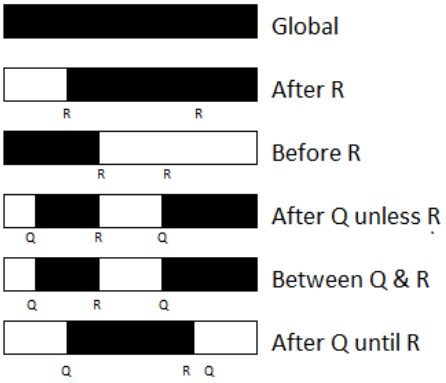
JML supports a lot of other features, e.g. ghost variables and model fields. Those are additional objects that only live in the specification and are needed for more complex specifications. A full list of the features of JML can be found in the *JML reference Manual* [LPC+08].

A method can be declared with the keyword `pure` which means essentially the same as adding `assignable nothing` and `diverges false` to every specification case of the method. Pure methods can be used in expressions within the specification as they are side effect free. On the top level temporaljmlc uses the temporal property specification scopes displayed in figure 4 .

5 Comparison With Other Verification Tools

The KeY Project is an interactive verification system that allows proving Java Card programs. But it also supports some features that are exclusive to Java such as static initialization or Strings. That means with some restrictions it's possible to prove full Java programs. Generics and threads however are for example not supported.

Figure 4: Temporal Property Specification Scopes in Temporaljmlc



Java Card is a limited version of Java that is designed to run on smart cards. Several advanced features such as generics are not available in Java Card. In KeY you prove the correctness of dynamic logic formulae. This is similar to proving Hoare triples. With Hoare triples you have a precondition, a postcondition and the program itself. You prove that each time the precondition is fulfilled before running of the program then afterwards the postcondition holds.

The Hoare triple $P\{A\}Q$ can be represented in dynamic logic by $P \rightarrow [A]Q$. In Dynamic Logic the equivalent to this triple is again a formula. KeY with trace semantics is more expressive than Hoare logic because with KeY several traces can be used in the same sequent. For an example look at Section 3.3 .

The input for every verification is a formal specification. In KeY there are two possible ways of providing a specification. Providing the input is possible via the Java Modeling Language (JML) or directly in Java Dynamic Logic (JavaDL). If you already have a specification in JML you just have to select the method contracts you want to verify via a menu in the KeY program. In this dialog it is also possible choose which invariants to use for the verification. KeY then automatically generates the proof obligation.

KeY uses a sequent calculus for the generation of proofs. In the KeY standalone tool sequents are manipulated automatically as well as by hand. There are branching rules that lead to two open sequents that need to be closed. KeY keeps track of all changes to the sequent by managing a proof tree. This tree is graphically displayed like the folder structure in a file browser. Branches can be folded and unfolded to get a better overview. Closed branches are displayed in green.

Temporal logic has been used before in proof systems, for example in *temporaljmlc* [HL10] which is based on the logic *temporalJML* [TH02]. The use of the temporal operators in *temporaljmlc* is however more restricted. Furthermore the prover only does run time checking. *The Spin Model Checker* [Hol03] allows to give specifications in linear temporal logic is however restricted to finite models and data structures.

Table 1: Syntax of TemporalJML and temporaljmlc

temporalJML between method1 terminates method2 called x $>$ 0
temporaljmlc \between \call(method1); \terminates(method2) x $>$ 0

5.1 Temporaljmlc

Temporaljmlc [HL10] is a runtime assertion checking tool developed at the University of Central Floria, Orlando, FL, USA. It is based on a specification language called temporalJML [TH02]. Its goal is to simplify the specification of temporal properties. In contrast to javaDTL as used in the KeY verifier temporaljmlc restricts the possible temporal properties that can be expressed. The temporal property specification scopes can be expressed in javaDTL as shown in figure 4 .

5.1.1 States and Traces

Assume you have a class containing at least the methods *method1* and *method2*. Now you consider a single program run that involves various arbitrarily nested program calls. At various points during execution such as the start and the end of methods a snapshot of all variable values, the so called *state* (Def. 6), is taken. As Java is deterministic the *states* can be linearly ordered. This sequence of states is called *trace* (Def. 9) .

With the help of temporalJML you can easily define temporal properties about traces. In this example we want to express that each time *method1* finishes without an exception the value x has to stay positive until *method2* is called. This can be done as shown in table 5.1.1 . There are minor differences between the syntax of temporalJML and temporaljmlc.

The two programs TemporalJMLExample1 3 and TemporalJMLExample2 4 both increase x in steps of 1 from -1 to 2. In the first example *method1()* uses a recursive call to itself. The second example uses a while loop within the main method to repeatedly call *method1()*.

In temporaljmlc a trace consists of the visible states of a program. Visible states as described in the *JML Reference Manual* [LPC+08] are the entry and end points of methods also including intermediate helper methods.

5.1.2 Using Scopes and Specification Patterns

A scope is a subset of the states within the trace. It is used for specifying temporal properties that only have to hold for part of the program, namely the subtrace that is formed by the states of the scope. The following scopes are used in temporaljmlc[TH02]:

global throughout the entire program's execution

after the execution after a given state or event

```
public class TemporalJMLExample1 {
int x;
    /*@ \between \call(method1);
      @ \terminates(method2) x > 0 @*/
    public static void main(String[] args) {
        x = -1;
        method1();
        method2();
    }
    void method1(){
        if(x <= 1){
            x++;
            method1();
        }
    }
    void method2(){
        x = -1;
    }
}
```

Listing 3: In this example the program is **not** correct in respect to the specification

```
public class TemporalJMLExample2 {
int x;
    /*@ \between \call(method1);
      @ \terminates(method2) x > 0 @*/
    public static void main(String[] args) {
        x = -1;
        while(x <= 1){
            method1();
        }
        method2();
    }
    void method1(){
        x++;
    }
    void method2(){
        x = -1;
    }
}
```

Listing 4: In this example the program is correct in respect to the specification

before the execution up to a given state or event

between any part of the execution between two designated state or event

after-until the execution after a given state or event until another state or event or throughout the rest of the program if there is no subsequent occurrence of that state or event

In *Extending jml specifications with temporal logic* [TH02] the following definition has been given: "A specification pattern is a language independent set of commonly used specification constructs for finite-state verification ..." . A specification pattern for temporal formulae is a template which is instantiated with non-temporal expressions. Most temporal properties occuring within the verification process can be specified in a formal language closely related to the English one by using these patterns. The specification can then be expressed in linear temporal logic. Complicated, recurring constructs in LTL like in table 2 are hidden from the specifier who only needs to know the semantics of the patterns but not the translation to LTL. This makes the specification easier and less error-prone. A downside of this approach is that not all LTL-formulae can be expressed with the specification patterns temporaljmlc uses. For example the LTL-formula $\Box\Diamond P$ expressing that P holds in infinitely many states cannot be expressed with the specification patterns temporaljmlc uses.

The equivalent LTL formulae of different existence specification patterns are shown in table 2. The *Global* scope is the simplest of all as it just refers to the whole program run. The expression *After* Q *exists* P means that if event Q happens, then at some time after that the state formula P has to get true. In LTL this can be in the way that $\neg Q$ has to be true until a state in which Q and $\Diamond P$ holds. Because Q may never become true the weak-until operator is used.

To express the fact that a state formula P becomes true at some point before an event R happens can be expressed straightforward as *Before* R *exists* P. If P has become true before R this means that there has to be a state where $P \wedge\neg R$ holds before it is allowed to have a state in with R evaluates to true. In line 3 of table 2 the LTL formula that expresses this pattern can be found. The rest of the table also shows the correspondence of the other existence patterns to LTL.

Specification pattern can be devided into *Occurence Patterns* and *Ordering patterns*. There are the following Occurence Patterns:

- universal
- existence
- absence
- bounded existence

Especially for the specification of protocols it is useful to instantiate one of the *Ordering patterns* which are listed as follows:

- precedence
- response

Table 2: temporaljml scopes in LTL with existence patterns

Global exists P	$\Diamond$(P)
After Q exists P	($\neg$ Q) **W** (Q $\wedge$ $\Diamond$P)
Before R exists P	($\neg$ R) **W** (P $\wedge\neg$ R)
After Q unless R exists P	$\Box$ ($\neg$ Q)$\vee$ $\Diamond$(Q $\wedge\neg$ R $\Rightarrow$ ($\neg$ R **W**(P$\wedge\neg$ R)))
Between Q $\wedge$ R exists P	$\Box$ ($\neg$ Q)$\vee$ $\Diamond$(Q $\wedge\neg$ R $\Rightarrow$ ($\neg$ R **W**(P$\wedge\neg$ R)))
After Q until R exists P	$\Box$ ($\neg$ Q)$\vee$ $\Diamond$(Q $\wedge\neg R \Rightarrow$ ($\neg$ R **U**(P$\wedge\neg$ R)))

5.1.3 Comparison with KeY

In temporaljmlc a trace consists of the visible states of a program. This approach is more detailed than the handling in KeY without dynamic trace logic that considers only the pre- and post-state. (*The KeY Approach* [BHS07], 8.2.1). However KeY with dynamic trace logic is much more fine-grained namely on the level of single assignments.

For verifications in KeY with dynamic trace logic it is best to reduce the number of assignments to global fields to a minimum by using local variables because then the number of state transitions decreases and the proofing process becomes easier. If possible it is good to keep all the fields of an object in a consistent state all the time. This is almost always possible when you only consider the visible state as in temporaljmlc but not achievable when there are mutual dependencies on the fields and you consider every assignment as state transition. The best workaround is to have a boolean field or JML model field, let's call it *consistent* that is set to `true` if the object state is consistent. For some property p that has to hold in a consistent state you can then write the DTL formula :

$$\Box(\text{consistent} \Longrightarrow \text{p})$$

5.1.4 Semantics of TemporalJMLC

Table 2 shows how formulae using temporalJML scopes and the existential property pattern can be translated into DTL formulae. Note that there isn't exact equivalence between these formulae but only analogly because Temporaljmlc and KeY with dynamic temporal logic use different models for the trace of a program.

It is apparent that the temporaljmlc scopes are more human readable. LTL on which DTL is based is described as "overly mathematical for the average programmer". However the downside of Temporaljmlc's approach is its restricted expressiveness. Trace properties define the temporal behavior within Property specification Scopes that are colored black in figure 4 . These trace properties are: `\never`, `\always`, `\eventually` and `\atmost`. Contrarily in KeY with DTL the temporal operators can be arbitrarily nested even together with quantifiers. This gives the opportunity to write two-state-invariants as shown in section 9.3 . Formal methods as in KeY are used in cases where correctness of the program is vital. In many branches of computer science where the cost benefit ratio is more important traditional testing is being used. Programmers who work with formal methods such as verification have a mathematical knowl-

edge that is above average. So it might be fine if the specification is a bit less intuitive but much more expressive.

Temporaljmlc and KeY DTL have different approaches to deal with the problem that almost all decision problems dealing with verification are undecidable. Although these problems are in general undecidable, many real world instances are. So the goal for every verifier is to be able to handle a large fraction of the possible problems in a reasonable amount of time. Often one also has to deal with enormous state spaces what is called the state space explosion problem. Temporaljmlc is a runtime checker so it only has to deal with one concrete execution trace at once. For the checking of the temporal properties an finite automaton is automatically generated and updated during the run of the program with some test data input. In contrast to simple assertions in the source code the program has to be completely executed before Temporaljmlc can decide whether the temporal formula is fulfilled. This is because a trace property containing `\never` or `\always` could become invalid in the last moment in the program execution.

In spite of that the needed computation time is usually not a problem and also the process does not need human interaction. The main problem is the same as with testing: finding good test data so that also border cases are considered to reveal hidden bugs. The advantage to just testing the input/output behavior is that with temporaljmlc bugs can be found that do not manifest in wrong output for the test cases. This tool may also help to detect wrong assumption about the program early and help to spot bugs that may even come from the specification. However temporaljmlc can give no guarantees about the correctness because exhaustive testing of all inputs is not feasible.

KeY with dynamic temporal logic follows the approach of symbolic execution. This way very large or even an infinite amount of different possible inputs can be handled at the same time. If the execution comes to a branch point the proof has to be split into two independent proof branches. To deal with the potentially unmanageable big amount of data KeY uses invariant rules and heuristics that avoid too much branching. Nevertheless the state space explosion is a major problem for KeY. Manual interaction may be needed for steps of the proof like instantiation of quantifiers or decisions whether to unroll the program or the temporal operator. When the verification of a temporal property is done you have a higher lever of confidence for the correctness of your program compared to temporaljmlc.

5.2 The Spin Model Checker

For a verification task in the Spin Model Checker [Hol03] at first a simplified model of the software has to be constructed. The Spin Model Checker was in the first place developed for the verification of communication protocols. It can however be used for other purposes as long as the state space in the models is small enough.

The properties of a system can be formulated as linear temporal logic formulae (LTL) in Spin. It has support for concurrent processes and can even prove the correctness LTL formulae regardless of the interleaving. Spin does an exhaustive search on the state space to perform the verification.

The small state space makes it easier to generate a proof compared to KeY but the model that is verified is much more distant to the actual implementation

in the source code. In Spin the variables for integers are only one byte wide and you usually can only handle only a relatively small number of different inputs before you get a problem with the state space explosion. When you have successfully proven an LTL formula in Spin this does not have to be the case for your real implementation because of oversimplification or mismatches between the model and the source code. The specification in KeY is as close as possible to the source code because it is defined in JML. The KeY Version that supports dynamic temporal logic will stick with this specification language as input. To be able to use JML an extension to it has been developed that is presented in section 7.3 .

6 Taclets

Most rules in KeY are defined by so called taclets. They define how the sequent can be legally manipulated. There are a lot of taclets that come with KeY for defining the rules for JavaDL but it is also possible to define your own additional taclets. Taclets are described in *Verification of Object-Oriented Software - The KeY Approach* [BHS07], Chapter 4. The taclet language is kept rather simple. This way the writing of taclets is less prone to errors than hard-coding the rules in Java. However this way the number of taclets becomes relatively big.

6.1 Purpose of Taclets

Taclets are designed to allow automation of many proof tasks. They are rewrite rules that can simplify sequents and symbolically execute programs. Automation is achieved by global strategies that prioritize the different possible taclet applications. The rule with the highest priority is then applied automatically. It is possible to tune the global strategies in the settings of KeY to adapt it to different proof tasks.

There are also taclets that require user interaction such as rules that needs an instantiation of a quantifier. In this case taclets are used to describe how the interaction is done and what its effect on the sequent is. Taclets are designed to keep the user interaction clear and simple by limiting the complexity of a single rule.

Each taclet is stored in a seperate file to keep things organized for the person designing them. The set of the taclet serves as axioms for the proofing process. However KeY offers the possibility of proofing that a specific taclet can be deduced from another set of rules. For advanced users of KeY it is also possible to create user defined taclets.

6.2 Syntax and Semantics of the Taclet Language

The syntax of taclets is based on an ASCII representation in a text file. Rewrite rules are often given in the form of a fraction bar with a pattern for a sequent on which it can be applied at the bottom and the resulting sequent or sequents at the top. Whereas this form is easy to read, the ASCII format used in the taclet language has the advantage, that it can be written using a simple text editor. Keywords are identified by starting with a backslash (\). Top level keywords are followed by curly braces({}). The most important ones are listed here:

\sorts In the scope of this operator new types can be introduced. For example by `\generic G` a type is created that can stand for an arbitrary type. You can then later define variables or terms of that type.

\schemaVariables Schema variables are placeholders for different kind of constructs such as terms, formulae, programs or modal operators. Within this scope you can for example declare a formula by `\forumula phi;`, a term of integer type by `\term integer a;`, a variable of the generic type G by `\variables G x;` or an arbitrary Java expression by defining `\program expression #e;`.

\rules Here the rewriting rules can be defined. A rule starts with the name of the rule followed by the formal description what the rule does enclosed in curly braces. The underlying principle of the rules is pattern matching with the help of schema variables. The left and the right hand side of the sequent is divided by the symbol ==>. For example the rule `impRight { \find(==> phi -> psi) \replacewith(phi ==> psi)}` defines the rule that looks for a an implication on the right hand side of the sequent and moves the left part of it to the left hand side. The keyword `\find` defines the search pattern. The matching part of the sequent is replaced by the sequent that follows after `\replacewith`. In case of a positive match of the sequent defined after `\find` the sequent following the keyword `\add` is also added. With `\heuristics` followed by a heuristic name you can define which heuristic is used for the automatic execution. How the different heuristics are handled is however hard-coded in KeY.

```
\schemaVariables {
  \formula phi;
}

\rules{

  unwindBox {
    \find( \[[{.. ...}\]](\box phi))
    \sameUpdateLevel
    \replacewith(\[[{.. ...}\]](\wnext\box phi) & \[[{..
        ...}\]](phi))
    \heuristics(unwind_temporal)
  };

}
```

Listing 5: unwindBox Taclet

6.3 Taclets for Trace Semantics

For adding the trace semantics to the logic a lot of new taclets have to be added.

For the temporal operators *Box*, *Diamond*, *Until*, *WeakUntil* and *Release* taclets for unwinding rules have to be added. This unwindinding is similar to the unwinding of loops with the rule

$$\text{loopUnwind}\ \frac{\Rightarrow \langle \pi \text{ if } (e) \ \{\ p \text{ while } (e)\ p\ \}\ \omega \rangle \phi}{\Rightarrow \langle \pi \text{ while } (e)\ p\ \omega \rangle \phi}$$

which is described in *Verification of Object-Oriented Software - The KeY Approach* [BHS07], 3.6.4. However in the case of the temporal unroll operators not the program is unrolled but the temporal operator itself instead. For example the *unwindBox* rule

$$\text{unwindBox}\ \frac{\Gamma \vdash \mathcal{U}([\![\pi]\!] \bullet \Box\phi \wedge [\![\pi]\!]\phi), \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\ \Box\ \phi, \Delta}$$

can be written in the ASCII-style taclet language as shown in Listing 5.

The `\find` clause in the taclet causes KeY to look for an arbitrary program modality followed by $\Box\phi$ where ϕ is a formula. The box operator is then unwinded by replacing the matched expression with the expression following the `\replacewith` keyword. A formula is a boolean expression with the possible values *TRUE* and *FALSE*. It has to be differentiated from the Java type boolean. The focus is then set the found occurrence of this pattern. Adding the state condition `\sameUpdateLevel` prevents the program [[{.....}]] from occurring within phi.

7 Generating Proof Obligations from JML

The specification of a class is defined in JML annotations within the Java source code files. In addition to the well known JML constructs described in the *JML Reference Manual* [LPC+08] it may also contain the newly introduced temporal extensions described in section 4 .

Proving the validity of the whole given specification at once is often too complicated. For that reason several so called proof obligations are created. A proof obligation is a JavaDTL formula that is used for the proof process in KeY. A user interface allows to select aspects that one wants to verify, e.g. that a method obeys its contract or that invariants are conserved. After this choice the appropriate JavaDTL formulae are automatically generated. When all the separate proof obligations have been successfully completed it is verified that the source code is correct in respect to the specification.

7.1 Proof Obligations in Non-temporal KeY

To explain what changes have to be done for generation of temporal proof obligations A short overview on how non-temporal proof obligations are handled in the contract concept of KeY 1.0 is given in this subsection. The contract concept is used for method contracts. This concept can be adapted to traces that result from a single method call as will be shown in section 7.2 . A more detailed description can be found in *Chapter 8, The KeY Approach* [BHS07]. The proof obligation templates taken from there are also shown in table 4 .

The tables 3 and 4 use various abbreviations that are explained here. They are important for the understanding of how proof obligations of the KeY 1.0 contract concept have to be accommodate to the usage of temporal clauses.

op an operation, i.e. a series of atomic operation steps, usually a method execution, of one of the considered classes.

opct an operation contract for *op* with
opct = (Pre, Post, Mod, Termin) with the precondition *Pre*, the post-condition *Post*, the modifies clause *Mod* and the term defining when the operation has to terminate

I, Assumed, Ensured subsets of the invariants in Inv_{Spec}, with I being the set of all static invariants, *Assumed* and *Ensured* are subsets of invariants and can be selected in the user interface when proving that invariants are preserved by an operation. Invariants in *Assumed* are added to the precondition, invariants in *Ensured* to the postcondition. To show that all invariants are preserved by an operation, every invariant has to be at least in one closed prove in the *Ensured* set for that method.

Conj$_{\mathbf{F}}$ conjunction of the formulae contained in set F. It is used for $\text{Conj}_{\text{Assumed}}$ and $\text{Conj}_{\text{Ensured}}$ to handle multiple assumed Invariants and Invariants on the ensured side of the sequent.

Disj$_{\mathbf{Pre}}$ disjunction of all preconditions for *op*. Here the disjunction is used because by definition the semantic of several preconditions is that at least one have to hold.

$\Phi_{\mathbf{init}}$ In the initial state all variables still have their

default values. In this the boolean variable $\langle C.classPrepared \rangle$ is set to $FALSE$ for all classes C. The conjunction of $\langle C.classPrepared \rangle = FALSE$ for all C is described as Φ_{init}. The boolean variable $\langle C.classInitialised \rangle$ is $TRUE$ for all classes C in that case. The purpose of $\langle C.classInitialised \rangle$ is to have a way to specify that static invariants have to hold only after finishing of the initialization.

$\mathcal{V}$ an anonymising update $\mathcal{V}$ is used to prove that a formula is valid regardless of the context. It is used for example in the invariant rules in KeY. A formal descripition of anonymising updates is given in *Definition 3.59 in The KeY Approach* [BHS07].

ValidCall$_{\mathbf{op}}$ this formula defines that the parameters of a method call have the correct variable type and that they fulfill the precondition of an operation contract.

InitInv(I) This template generates the obligation to proof that in the initial state all the invariants hold.

PreservesInv(op; Assumed; Ensured) Here it is shown that assuming all the Invariants in *Assumed* at least the invariants in the set *Ensured* have to hold afterwards if the operation terminates.

PreservesOwnInv(op; Assumed) In this special case of *PreservesInv* all the instance variables of a class are ensured.

EnsuresPost(opct; Assumed) The purpose of this template is to generate a DTL-formula that states that a method obeys an operation contract.

This formula states that when in the prestate the precondition *Pre*, the assumed invariants $Conj_{Assumed}$ as well as $\text{ValidCall}_{\text{op}}$holds then in the poststate the postcondition *Post* has to hold. There is a distinction in this template between proving partial and total correctness which is determined by*Termin*

RespectsModifies(opct; Assumed) This pattern is for generating a DTL-formula that states that at most the variables described in *Mod* can change. To express this there is an anonymizing update $\mathcal{V}$ that undoes the changes to all fields mentioned in *Mod* and shows that the state is the same as before. If there a change to an other field were made then the states would differ afterwards. To express the equivalence of two states in DTL the anonymous method *anon()* is used. Two states S_1 and S_2 are equivalent if the termination of *anon()* started in S_1 implies the termination of *anon()* started in S_2.

Table 3: Non-temporal proof obligations templates for program correctness

Proof Obligation Template	Formula
InitInv(I)	$\Phi_{init} \rightarrow \text{Conj}_I$
PreservesInv(op; Assumed; Ensured)	$\text{Conj}_{\text{Assumed}} \wedge \text{Disj}_{\text{Pre}} \wedge \text{ValidCall}_{\text{op}}$ $\rightarrow [\text{Prg}_{\text{op}}()]\ \text{Conj}_{\text{Ensured}}$
PreservesOwnInv(op; Assumed)	$\text{Conj}_{\text{Assumed}} \wedge \text{Disj}_{\text{Pre}} \wedge \text{ValidCall}_{\text{op}}$ $\rightarrow [\text{Prg}_{\text{op}}()]\text{Conj}_{\text{I}}$
EnsuresPost(opct; Assumed)	$\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}}$ $\rightarrow \langle \text{Prg}_{\text{op}}() \rangle \text{Post}$ (if Termin = total) $\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}}$ $\rightarrow [\text{Prg}_{\text{op}}()]\text{Post}$ (if Termin = partial)
RespectsModifies(opct; Assumed)	$\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}}$ $\wedge\ \text{PreAxioms} \wedge \{\mathcal{V}\}\langle \text{anon}();\rangle \text{true}$ $\rightarrow [\text{Prg}_{\text{op}}()]\{\mathcal{V}\}^{@\text{pre}}\langle \text{anon}();\rangle \text{true}$

7.2 Changes Needed for Trace-based Proof Obligations

To the operation contract(optc) the additional term *Temporal* has to be added to handle the trace-based specification. So now the tuple *opct* is defined as: *opct* = (Pre, Post, Mod, Termin, Temporal). In the following I will show which additional proof obligation templates are needed.

For the KeY version with dynamic trace logic the additional proof obligation for the temporal-clause has to be generated. The corresponding template is shown in table 4 along with the already existing ones. The proof obligations known from the previous KeY version without dynamic logic remain unchanged and simply ignore the *temporal* clause.

It is possible to replace occurrences of $[\text{Prg}_{\text{op}}()]$ and $\langle \text{Prg}_{\text{op}}() \rangle$by $[\![\text{Prg}_{\text{op}}()]\!]$ followed by a temporal formula. So for example the Formula for EnsuresPost

can be written as

$$\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}} \rightarrow [\![\text{Prg}_{\text{op}}()]\!]\Box(\bullet\text{false} \rightarrow \text{Post})$$

in the case where total correctness is demanded and as

$$\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}} \rightarrow [\![\text{Prg}_{\text{op}}()]\!]\Diamond(\bullet\text{false} \wedge \text{Post})$$

in the case where only partial correctness is needed. These proof obligations can after this transformation be combined with the EnsuresTemporal formula to

$$\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}} \rightarrow [\![\text{Prg}_{\text{op}}()]\!]((\bullet\text{false} \rightarrow \text{Post}) \wedge \text{Temporal})$$

and

$$\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}} \rightarrow [\![\text{Prg}_{\text{op}}()]\!](\Diamond(\bullet\text{false} \wedge \text{Post}) \wedge \text{Temporal})$$

respectively. However it is better to keep the proof obligations separate in order to split up a big proof into several smaller proofs for easier verification. In the example above the top level operator is $\wedge$. So most likely the proof would split up after a *pullOutBelowTrace rule* followed by a *andRight rule* anyway.

Keeping the EnsuresPost proof obligation along with the EnsuresTemporal one might introduce duplicate work to be done in the verification Process. The ensures clause must contain all the information other methods need about the result of the method. It is better to keep dynamic trace logic out of the verification process for ensuring the post condition. Dynamic trace logic is very expressive but this also makes the proofing process more difficult as you have to decide whether to unroll the program or the temporal operator.

However after finishing the proof for EnsuresPost with JavaDL one might use the result for the EnsuresTemporal proof by adding $[\text{Prg}_{\text{op}}()]$ or $\langle\text{Prg}_{\text{op}}()\rangle$ to the left hand side of the EnsuresTemporal proof obligation template. That would lead in the case for total correctness to the following formula:

$$\begin{aligned}\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}} \ \wedge [\![\text{Prg}_{\text{op}}()]\!]\ (\bullet\text{false} \rightarrow \text{Post}) \\ \rightarrow [\![\text{Prg}_{\text{op}}()]\!]\text{Temporal}\end{aligned}$$

This might be helpful in some cases especially when you have encoded parts of the ensures clause again within the temporal clause. However the better approach is trying not to put too much into the temporal clause as proofing DTL formulae is difficult.

Table 4: New temporal proof obligations templates for program correctness

Proof Obligation Template	Formula
PreservesStrongInv(op; Assumed; Ensured)	$\text{Conj}_{\text{Assumed}} \wedge \text{Disj}_{\text{Pre}} \wedge \text{ValidCall}_{\text{op}} \rightarrow [\![\text{Prg}_{\text{op}}()]\!]\Box\text{Conj}_{\text{Ensured}}$
PreservesStrongOwnInv(op; Assumed)	$\text{Conj}_{\text{Assumed}} \wedge \text{Disj}_{\text{Pre}} \wedge \text{ValidCall}_{\text{op}} \rightarrow [\![\text{Prg}_{\text{op}}()]\!]\Box\text{Conj}_{\text{I}}$
EnsuresTemporal(opct; Assumed)	$\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}} \rightarrow [\![\text{Prg}_{\text{op}}()]\!]\text{Temporal}$

In KeY with dynamic temporal logic the additional modifier *strict* for invariants is added. Without the modifier strict the semantic of invariants in KeY is that they have to be preserved by all methods in regard to the observed-state semantics *Verification of Object-Oriented Software - The KeY Approach" [BHS07], 8.2.*. That means that if an invariant holds at the beginning of a method it has to hold at the end. However in between it might not hold for some time.

Invariants marked with the modifier *strict* however have to hold after every step of the execution of a method. This definition is even far more restrictive than the JML visible state semantics. A state is visible at the beginning and the end of methods. The main difference to the observed state semantics is that the start and endpoints of intermediate methods called during execution of a method are visible according to JML visible state semantics but hidden in observed state semantics. *The KeY Approach* [BHS07], 8.2.1

Proofing that strict invariants are preserved by a method is more difficult as dynamic temporal logic has to be used. However strict invariants are very useful with concurrent threads. The strict invariants hold even when another thread is currently executing a method that affects the class attributes.

7.3 Integration of Trace-Specification in JML

In order to make use of the newly introduced trace operators in a JML specification the JML language has to be extended. This is done by introducing a new temporal clause with the keyword `temporal`. The temporal clause is defined within a behavior case on the same level as `requires` and `ensures` clauses. Its syntax is closely related to linear temporal logic and is defined in Table 5.

The semantic of the temporal clause is that the expression that may contain temporal operators has to be true. The meaning of the temporal operators is as in linear temporal logic were the state transitions are assignment to object fields in the program. A `temporal` clause can be defined in addition to other clauses like `requires`, `ensures`, `assignable` or `diverges`. In the normal behavior case the `temporal` has to be fulfilled in addition to the other clauses that are defined in the *JML Reference Manual* [LPC+08]. Abrupt termination that is defined in a exceptional behavior case is considered to be non-termination for the interpretation of the temporal clause. Several temporal clauses are allowed and considered as a conjunction semantically.

7.4 Example of Generating a Temporal Proof Obligation

This subsection explains how temporal proof obligations are automatically generated from a specification in JML. Later in this document in section 9.6 the same example is used to show how a proof for this problem can be generated. In listing 6 you can see the definition source code of the class *Bank* that contains a method contract in JML. To show the validity of the ensures clauses the same approach as in KeY1.0 is used without considering trace semantics. The *EnsuresPost* template is used to create the proof obligations. The RespectsModifies proof obligations are also handled in the same way as before because the semantics of the *modifies clause* has not been changed.

The *temporal* clause is new to KeY with trace semantics. For proving that the program conforms to the specification expressed in this clause the EnsuresTemporal template is used to generate a new sequent in KeY. When the proof tree

of that sequent can be closed the temporal clause of that method contract has been verified. For this example JML specification displayed in listing 6 . The formula displayed in table 6 is generated by the proof obligation template *EnsuresTemporal* which is shown in the overview over the non-temporal templates in table 3 . In this example no invariants are needed for the proof, so $\text{Conj}_{\text{Assumed}}$ is empty. *Pre* is as explained before the conjuction of the preconditions that are taken from the JML specification of the corresponding method contract. The $\text{ValidCall}_{\text{op}}$ term contains subterms that ensure that the method has been called with arguments that are valid for that specific method contract. This term appearing in table 6 is rather technical, but it can be expanded in the same way as in KeY1.0. For better readability *sender* is abbreviated by *s* and *balance* by *b*.

The requires clauses are added to the JavaDTL sequent on the left hand side as prerequisites. This example also shows how the $\backslash old$ operator is handled. If some variable or object within the specification of the temporal clause is under the $\backslash old$ operator a new existentially quantified logical variable is introduced; in this example it is named *s.b@old*. Its purpose is to "save" a value in the first state of the trace to allow comparison to it in other states. The term $s.b@old = s.b$ is added to the formula to enforce this. The trick is that the existentially quantified variable *s.b@old* is immutable, i.e. its value is independent of the program state, whereas the value of *s.b* is state dependent. With the help of *s.b@old* the rest of the DTL formula can be formulated quite straight forward. The second temporal clause is constructed in an analog way. It's possible to do this process of generating a sequent from a JML specification automatically with the help of taclets. However it is not implemented in the prototypical implementation yet.

8 Changes from KeY 1.0 to KeY with Trace Logic

The extension of KeY to support verification of specified temporal formula has been designed to be backwards compatible with KeY 1.0. Therefor the *temporal* clauses in method contracts and the use of *strict* invariants (see section 7.2) are optional. The idea behind this is that generating proofs for temporal method contracts is more difficult than proofs containing without temporal logic. Therefor the non-temporal parts of the specification are proofed separately from the temporal part. Examples of the usage of KeY with trace logic will be given in section 9 .

8.1 Invariants and Variants

The semantic of both class and instance invariants has not changed. They still only have to hold at the start and the end of methods, but not in between. The Reason for this is that otherwise it would be almost impossible to do changes to data structures mentioned in invariants because often inconsistent temporal states are needed when a bigger change has to be done. On the other hand the additional modifier *strict* (see section 7.2) has been introduced. Its semantic is that the invariant with this modifier have to hold in every state. With a strict invariant you can for example specify a range for a variable that always have to hold. The advantage of such strict invariants is that even concurrent threads or

```
public class Bank {
  /*@ public normal_behavior
    @ requires sender != null;
    @ requires receiver != null;
    @ requires amount >= 0;
    @ assignable sender.balance, receiver.balance
    @ ensures sender.balance ==
    @     \old(sender.balance) - amount
    @ ensures receiver.balance ==
    @     \old(receiver.balance) + amount
    @ temporal sender.balance == \old(sender.balance)
    @     \until (\box sender.balance
    @     == \old(sender.balance) - amount)
    @ temporal sender.balance == \old(sender.balance)
    @     \until (\box sender.balance ==
    @     \old(sender.balance) + amount)
    @*/
    public void transfer(Account sender, Account receiver
        , int amount) {
    sender.balance = sender.balance - amount;
    receiver.balance = receiver.balance + amount;
    }
}
```

Listing 6: Bank.java

Figure 5: Example of the use of the \old operator in JML notation

```
\box(counter≥\old(counter))
```

processes can rely on them.

Variants are a tool used in verification of code containing loops. While invariants encode what stays the same throughout the loop body, variants encode the things that change. And explaination on how variants are defined in the *JML Reference Manual* [LPC+08] and in *Dynamic Trace Logic* [BB13] is followed by a description of how variants are implemented in KeY with trace semantics. There are some small but important differences.

A JML variant is a `long` or `int` expression that has to be non-negative after every loop iteration. It also has to decrease by at least one every loop iteration. These two facts about a variant can be used to prove termination. Several variants may be used.

In dynamic trace logic a variant V(x) is a DTL formula with a free logical mathematical integer variable. Variants can be entered interactively in KeY or defined in the JML comments of the source files.

8.2 Refering to the Inital and Return Values

In JML there is an operator defined called `\old`. It is already used in non-temporal method contracts of KeY. This operator can also be used in the case of temporal method contracts to give the user of KeY a convenient way to refer to the value of a variable at the first state in the trace. The semantic of `\old(exp)` in this case is that the expression `exp` is evaluated in the prestate before entering the method. All sub-expressions of `exp` are also evaluated in the prestate. The `\old` operator for the EnsuresTemporal proof obligation can be implemented for the temporal clause in the same way as in Java DL without trace semantics

Consider the example 5 of a definition of an DTL formula using the operator `\old`. This formula states that the variable `counter` is in every step at least the initial value counter. Decreasing the counter is allowed but not below the initial value. This DTL formula leads to the following proof obligation:

$$\begin{aligned} &\text{Conj}_{\text{Assumed}} \wedge \text{Pre} \wedge \text{ValidCall}_{\text{op}} \rightarrow \\ &\{\_\text{old1}:=\text{self\_Testclass.counter}\}\ [\![\text{Prg}_{\text{op}}()]\!]\ \Box(\text{counter} \geq \_\text{old1}) \end{aligned}$$

_old1 is a newly introduced program variable that is assigned to the initial value of counter before the method call. Because it is new it does not appear within the method body $\text{Prg}_{\text{op}}()$. Therefore it stays the same throughout the program trace and can be used in the DTL formula to refer to the initial value.

The `\result` operator is not allowed in the temporal clause. The reason is that in the case of non-termination the result value would be undefined. Properties about the value of result have to be defined within the ensures clause.

temporal-clause	::= *temporal-keyword temporal-formula*
temporal-keyword	::= `temporal`
temporal-formula	::= *temporal-keyword temporal-spec-expression*
temporal-spec-expression	::= *unary-temporal-expression*
	`|` *binary-temporal-expression* `|` *spec-expression*
unary-temporal-expression	::= *(unary-temporal-operator*
	temporal-spec-expression)
binary-temporal-expression	::= *(temporal-spec-expression*
	binary-temporal-operator temporal-spec-expression)
unary-temporal-operator	::= `\box| \diamond| \wnext| \stnext`
binary-temporal-operator	::= `\until| \wuntil| \release`

Table 5: JML extension described in bachus-naur-form

Table 6: instantiated proof obligation template for the bank example
$s \neq null \ \wedge \ receiver \neq null \ \wedge \ amount \geq 0 \rightarrow [[\mathrm{Prg}_{\mathrm{op}}()]] \ \exists \, int \, s.b@old;$
$(s.b@old = s.b \ \wedge \ (s.b = s.b@old \quad \backslash until \, \backslash box \ (s.b = s.b@old - amount)))$

9 Proving the Validity of Trace Formulae Interactively with the Help of KeY

All example problems in this section are defined directly in a text file with the suffix *.key* because the parsing of JML annotations containing temporal specifications is not yet implemented. A sketch of how generating the proof tasks from JML would work is however given in section 7.4 . With the current development version of KeY there is quite some interaction needed. This pattern of interactively selected rules repeats itself throughout all the examples.

The proof obligation of *Minimal Example* 9.1 is very basic and was the first temporal problem the author proved with the prototypical implementation of KeY with trace semantics. At the time of its writing there were no heuristics for the automatic execution of many temporal rules implemented yet. Therefore the most important temporal rules are explained with the help of this example.

The examples named *Counting Up* 9.2 and *Infinitely Counting Up* 9.3 show how two state invariants are handled in the case of a finite and an infinite trace. The special feature about *PaperExample* 9.4, which is taken from *Dynamic Trace Logic* [BB13], is that it has to be shown that a variable converges to a given value at some time during the execution of an infinite program. In the example*Multiply* 9.5 it is shown how JavaDTL can be used to express invariants in a temporal formula. Finally the *Bank* 9.6 example defines a problem that is close to some real case usage.

9.1 The Concept of Method-Frames

This is a minimal example that uses the trace semantics of KeY. Except for the *Diamond* operator this example is like dynamic logic without trace semantics. This very simple example demonstrates the proof process in detail. The java source code and the KeY file are shown in listings 7 and 8 . Many of the rule application are candidates for further automation. Especially for *assignment* rules containing *StrongNext*, *applyUpdateOnNontemporal* and *methodCallEmpty* rules there is no problem about automating them. However the *unroll* rules that roll out temporal operators can potentially be applied infinitely often. So by default these rules should not be applied automatically. But an option would be to add a checkbox to the GUI to let the user decide to unroll such rules without interaction similarly to the *unwindLoop* rule.

```
public class Assignment {
  public int var;

  public void set (int x) {
    var = x;
  }
}
```

Listing 7: Assignment Sourcecode

This proof uses method-frames [BHS07] (Chapter 3.6.5) a well as heap and store semantics [Wei11]. Method frames are the way KeY handles the replace-

```
\javaSource "source/";

\programVariables {
  Assignment a;
}

\problem {
  a != null ->
  \[[ {
    a.set(3);
  }
  \]]\diamond(a.var = 3)
}
```

Listing 8: Assignment KeYFile

ment of method calls with the actual implementation. In front of the method additional information is stored such as the class of the method called, the return type and where the return value of the method shall be stored if the method is not a void method.
In earlier versions of KeY object fields were modeled by non-rigid functions. In the current version the non-rigid function symbol *heap* is used to model the Java heap with its values of object fields. Non-rigid means that the interpretation may change in contrast to rigid function symbols as for example arithmetic operators. With the object o and the Field f the expression $o.f$ is represented by the term *heap(o, f)*. The advantage of this approach is that you can refer to the fields itself as they are not only function symbols. This is for example an advantage when you are dealing with dependencies between function symbols. For the following examples however the heap and store semantics make no difference. So you can just think of it as an alternative notation.

After method body expansion and some simplification steps the automatic prover stops here:

```
==>
 {x:=3}
   \[[{method-frame(source=set(int)@Assignment): {
          var=x;
        }
      }\]] (\diamond a.var = 3)
```

The *unrollDiamond* rule has been applied. For this proof we only have to consider the first part of the disjunction. The second one can be discarded:

```
==>
 {x:=3}
   (  \[[{method-frame(source=set(int)@Assignment): {
             var=x;
           }
         }\]] (\stnext \diamond a.var = 3)
    | \[[{method-frame(source=set(int)@Assignment): {
             var=x;
           }
         }\]] a.var = 3)
```

The assignment rule for *StrongNext* has been applied. The method body is now empty:

```
==>
 a = null,
 {x:=3}
   {heap:=store(heap, a, var, x)}
     \[[{method-frame(source=set(int)@Assignment,this=a):
          {}
        }\]] (\diamond a.var = 3)
```

The *methodCallEmpty* rule removes the method frame:

```
==>
 a = null,
 {heap:=store(heap, a, var, 3)}
   \[[{}\]] (\diamond a.var = 3)
```

The *unrollDiamond* rule has been applied again. Here we need the second part of the disjunction for the proof:

```
==>
 a = null,
 {heap:=store(heap, a, var, 3)}
   (  \[[{}\]] (\stnext \diamond a.var = 3)
    | \[[{}\]] a.var = 3)
```

The *applyUpdateOnNontemporal* rule has removed the empty program fragment. This is the last interactive step needed. For the rest of of the proof the store semantics described in *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction* [Wei11] are used.

9.2 Using Quantifiers in Trace Formulae

This exampleshows how you can proof formulae with quantifiers combined with LTL formulae. Here we want to state that in every state the following holds:

```
==>
 a = null,
 {heap:=store(heap, a, var, 3)}(a.var = 3)
```

`c.counter` will be increased by one if it is not the final state. A way is needed to talk about the value of `c.counter` in the next state. The newly introduced temporal operator *WeakNext* is needed for this task. However *WeakNext* cannot be applied directly on `c.counter` because `c.counter` is a program variable but the temporal operators only work on logic formulae. The next section shows another illustration of this difference and explains why there is even a difference between a boolean program variable and a logic formula. The solution to the problem we want to specify is to introduce a new existentially quantified variable `u` as seen in the listing below. The variable `u` has obviously to be equal to `c.counter` in every state. Within the *WeakNext* operator u represents the value of `c.counter` in the state before. `c.counter` in contrast to `u` is affected by the *WeakNext* operator so we achieve the wanted specification. In this example the integer field is incremented a fixed number of times. The termination makes the proof process simpler as no invariant rules are needed. A similar program with an infinite loop is shown in section 9.3. The obligation can be proven by repeatedly applying *unwindBox* rules. However termination is not a necessary requirement for a proof.

```
public class CountUp {
  public int counter;

  public int start(int max) {
    while(counter < max) {
      counter = counter + 1;
    }
  }
}
```

Listing 9: CountUp.java

For this proof *Loop treatment* in the *Proof Search Strategy* tab has been set to *Expand* because the loop in this example is finite and short enough for a reasonable proof size. This way the only needed manual interaction is the instantiation of existential quantifiers.

The following screenshot shows the complete proof tree in the state where the automatic proof stops. In this view all branches are expanded but intermediate proof steps are hidden for better clarity. You can see four open branches that are labeled with *OPEN GOAL* in red color and three closed branches that are indicated by a green folder. Unfortunately the default labels for branching rules in KeY are often not that meaningful therfor additional annotations to the screenshot in black font on white background have been added. The variables x and x_3 are boolean variables that were introduced automatically during the proof process.

By looking at the intermediate proof steps you can see that its value is equal to $counter < max$ in a specific state. The proof contains four times the

```
\javaSource "source/";

\programVariables {
  CountUp c;
}

\problem {
  c != null ->
  {c.counter := 0}
  \[[ {
    c.start(3);
  }
  \]] (\box(\exists int u; (c.counter = u & \wnext(c.
     counter = u + 1))))
}
```

Listing 10: CountUp.Key

rule *unwindBox*. This rule is called *R20* in *Dynamic Trace Logic* [BB13] and unwinds the trace operator by one step. The resulting formula contains an $\wedge$ on the right hand side of the sequent which leads to a branching caused by the rule *andRight* later on. The branches of these *andRight* rules are labeled with *Case 1* and *Case 2* respectively. In the branch *if x_3 true* this branching is not necessary because both `c.counter` and `c.max` have the value 3 which is a contradiction to *c.counter* $<$ *c.max*. This inequality on the left hand side of the sequent simplifies to *false* and thus the rule *closeFalse* can be applied.

Let's now look at the proof obligation that's labeled with *97:OPEN GOAL*. The sequent in this proof state is shown in listing 11 .

```
==>
 c = null,
 \exists int u;
   {max:=3 || heap:=store(heap, c, counter, 1)}
     \[[{method-frame(source=start(int)@CountUp,this=c):
         { {
               counter=counter+1;
           }while ( counter<max ) {
             counter=counter+1;
           }
         }
       }\]] (u = c.counter & (\wnext u = -1 + c.counter)
           )
```

Listing 11: state 97

The term *c* = *null* on the right hand side expresses the same as *c!= null* on the left hand side of the sequent. The case where the object `c` is `null` is handled in the branch labeled *Null Reference(c = null)* which closed automatically. The other term has the top level operator \exists. The proof stops here because

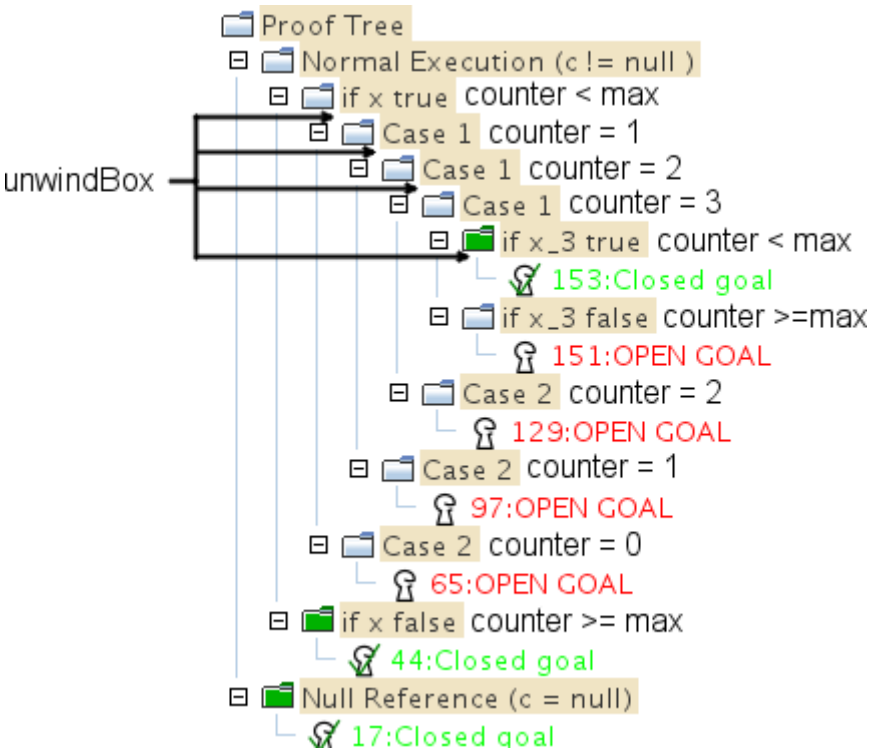


Figure 6: proof countUp

there aren't any rules left that can be applied automatically.

In the listing above you can see method-frames [BHS07] (Chapter 3) and the use of heap and store semantics [Wei11]. For the exact semantics you can look at the referenced sources but for this example it is enough to consider the parallel update as $\{max:=3||c.counter:=1\}$ and ignore the method-frame term. Inside the program modality you can see the while loop followed by the loop body. The loop body was placed in front of the while loop by a previous automatic application of the rule *unwindLoop*. To proceed manually you have to manually selected the $|$*exists*-term by a right click and select the rule *exRightHide*. In the window that will open u has to be instantiated with the value *1*. You have to choose this value for u because it's the only possibility to make the term $u = c.counter$ and thus the whole sequent true. The sequents in the proof states 65 and 129 are handled in the same way with the only difference that `c.counter` and the needed instantiation for u is *0* and *2* respectively. The sequent in the branch labeled *if x_3 false* is presented in listing 12 .

In contrast to the sequent considered before here the program in the program modality has disappeared because it has been completely executed symbolically. Nevertheless you have to instantiate u with *3* here. After these 4 instantiations the proof will close automatically pressing the *run* button.

```
==>
 c = null,
 \exists int u;
   {heap:=store(heap, c, counter, 3)}
     \[[{}\]] (u = c.counter & (\wnext u = -1 + c.counter
        ))
```

Listing 12: state 151

```
==>
 \[[{ {while ( true ) {
           counter=counter+1;
         }
       }
    }\]] (\box \exists int u; (u = c.counter & (\wnext u
       = -1 + c.counter)))
```

Listing 13: state 19

9.3 Validating Two-State Invariants for Non-Terminating Programs

This program counts up a counter to infinity. This example has the purpose to demonstrate how non-terminating programs can be handled. The only difference to the Section 9.2 is that the parameter `max` has been omitted and the loop now runs for ever with the loop condition `while(true){...}`

Loop treatment is set to *Invariant* in the *Proof Search Strategy* tab. The proof tree is shown in Figure 7

In proof step 19 you have to stop the automatic proof to choose the invariant rule instead of the *unwindBox* rule. The invariant and the variant are simply *true* and *0*. The (simplified) sequent looks like this when you have to do this invariant instantiation:

The branches *Invariant Initially Valid* and the *Use Case* close automatically. However in the *Body Preserves Invariant* branch some manual interactions are needed because the automatic prover seems to chose the wrong rules. Especially a rule labeled *One Step Simplification* > 1 that puts the *Exists* quantifier above the update *{heap:=anon(heap, allLocs, anon_heap_loop) || exc:=FALSE}* is causing problems. In proof step 80 the rule *exRightHide* is used with the instantiation `c.counter` for `u`. After this instantiation the *Body Preserves Invariant* branch can be close and thereby the whole proof.

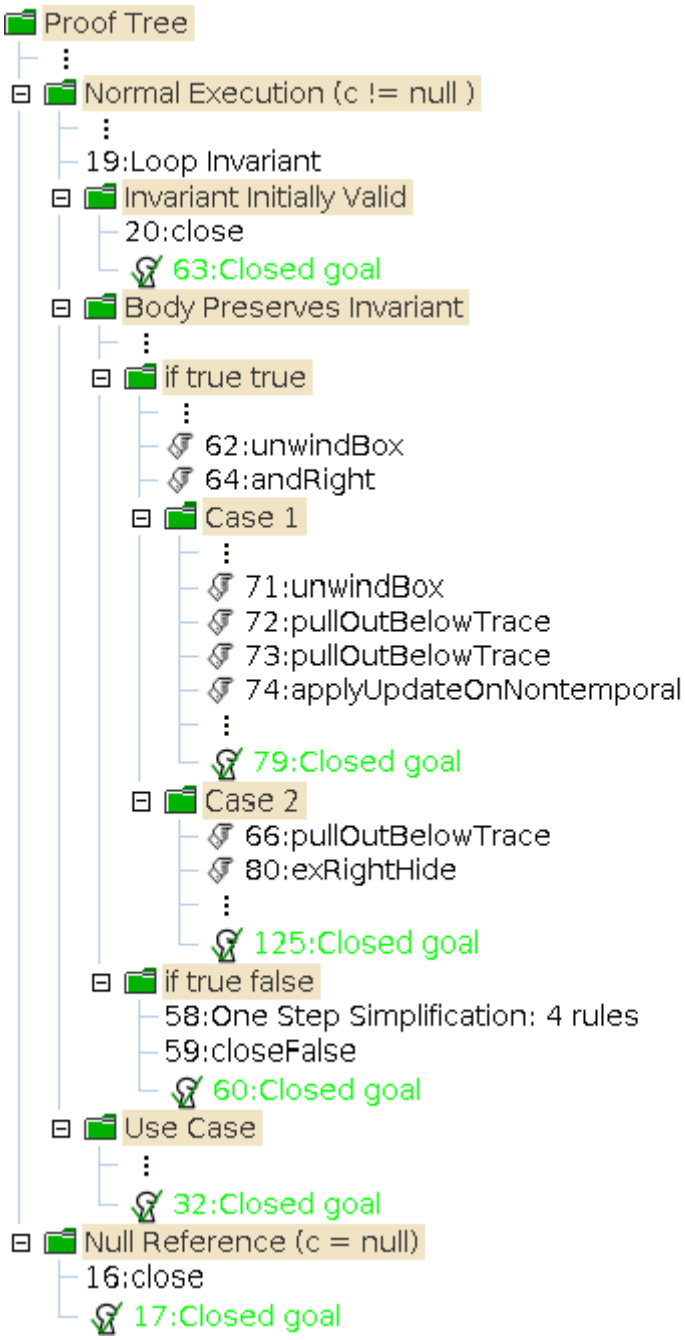


Figure 7: Proof CountUp

```
public class PaperExample {
  public int X;

  public int start() {
    while(true) {
      X = X - X/2;
    }
  }
}
```

Listing 14: PaperExample.java

9.4 Proving the Convergence of an Integer Sequence

In *Dynamic Trace Logic* [BB13] an example of proofing a trace semantic property for a program written in a simple while language is presented. In this chapter this example program will be presented again but written in Java. This program is special because it will run for ever. However as long as start value of `X` at the beginning is positive the program will end up in a state where `X==1` and stays forever at this value. The remarkable thing is that KeY with dynamic trace logic is able to proof this although the state in which `X` becomes `1` for the first time depends on the initial value.

```
\javaSource "source/";

\programVariables {
  PaperExample c;
}

\problem {
  c != null & wellFormed(heap) & c.X > 0 ->
  \[[ {
    c.start();
  }
  \]] (\diamond(\box(c.X = 1)))
}
```

Listing 15: PaperExample.Key

Loop treatment has been set to *Invariant* in the *Proof Search Strategy* tab. Temporal unwinding rules are applied automatically. The Proof tree is shown in the figure 8 .

For the proof you have to automatically execute proof steps until you arrive in step 15 at the following sequent.

Method frames, heap and store semantics and unnecessary parts of the sequent are omitted in this and the following listings for better clarity. Without being stopped, the automatically execution would now continue with an *unrollDiamond* rule. However for the proof you have to manually instantiate a *LoopInvariant* rule with the invariant *true* and the variant *c.X-1*. Later it will be possible to specify which invariants to apply in a JML comment so that this step can be performed automatically.

```
 c.X >= 1
==>
 \[[ {while ( true ) {
          c.X=c.X-c.X/2;
        }
      }
    }\]] (\diamond \box c.X = 1)
```

Listing 16: state 15

```
 c.X >= 1,
==>
 {c.X = 1)}
   \[[{ {while ( true ) {
            c.X=c.X-c.X/2;
          }
        }
      }\]] (\box c.X = 1),
```

Listing 17: state 51

The branches labeled *Positive Variant Implies Looping*, *Invariant Initially Valid* and *Body Preserves Invariant* close automatically. The automatic prover is able to do some simplification steps in the *Use Case* branch. Amongst them is an *unwindDiamond*-rule shortly followed by an *unwindBox* rule. Following the proof displayed in Fig.1 in *Dynamic Trace Logic* [BB13] we now prune the proof tree in front of this *unwindBox*-rule which leads to the following sequent in state 51:

The other term with $\circ\Diamond\Box$ *(c.X)* that resulted from the *unwindDiamond* rule is omitted in this listing because it is not needed in the proof. Now you have to apply the loop invariant rule with the invariant *c.X=1* and the variant *0*. Then the *Use Case* can be closed automatically.

The main difficulty in this proof is the instantiation of the needed loop invariants in the temporal invariant rules. It would be nice if KeY could find the right instantiations for the *Invariant*, the *Variant* and optionally the *Modifies* clause. However that would be at the most be possible for some special cases as the proving process is very complex and the existence of a proof undecidable. The next best alternative is to enable the user to specify the temporal invariant rule parameters within the JML annotations similarly to the invariant rule in the normal KeY version.

An additional challenge in the temporal case is the existence of three different temporal unwind rules for the operators $\Box$, $\Diamond$ and $\mathbf{U}$. The bigger challenge is however to determine when the temporal invariant rules have to be applied. As seen in this example for the same loop two different loop invariants have to be used; one time the $\Diamond$-invariant-rule and and one time the $\Box$-invariant-rule. In general any number of invariants with different parameters might be needed for the proving of a temporal property because the temporal operators can be nested arbitrarily.

Specifying all needed temporal invariants in front of a loop is not enough. In case several temporal invariant instantiations are specified for the same temporal operator and the same loop KeY has no clue to decide which specified parameter set to pick. In cases with only one parameter set for every temporal invariant case there is still the problem to decide whether to use the temporal unwind or the temporal invariant rule. In this example in step 15 the *Loop Invariant(Diamond)* rule has to be chosen however in step 48 the *unwindDiamond*-rule is the right choice. In both steps either rule is applicable.

An automatic choice seems to be difficult and giving an hint for this decision within the JML comments is a problem without knowing the proof structure. A solution to this problem has to be found because the specification of the invariants within the JML documentation is much more convenient than entering the invariant parameters interactively every time.

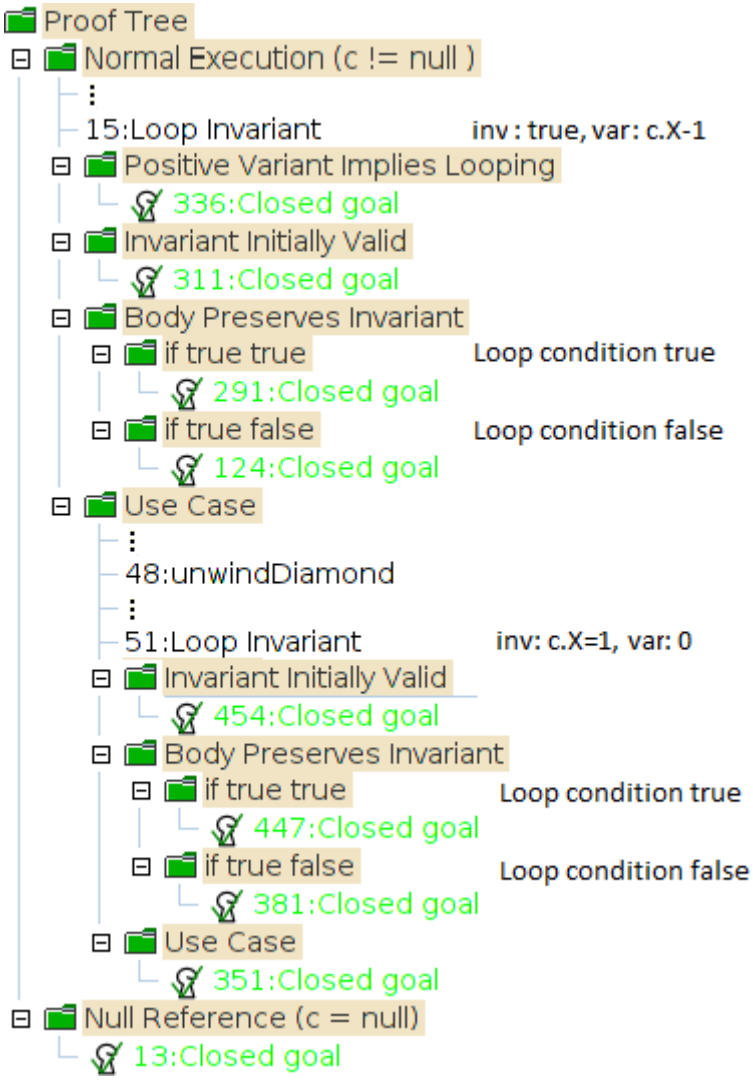


Figure 8: Proof PaperExample

9.5 Modelling Invariants as Trace Formulae

We have seen how to specify that a formula is true at some time during the program as well as always during the execution. In this example we want to specify that a formula B is only true in certain states, more specifically in states where formula A is true. You can achieve this in the following way: *Box ((A)→B))*. In this example A = *(∃ int u; (m.i = u ∧ WeakNext(m.i = u - 1)))*.

A describes the states after i is incremented. In these states A evaluates to *true* in all other states to *false*. A becomes true every time just after the command in *line 11* is executed. B is an invariant that holds after each iteration. From this example you see that it is a bit tricky to specify the states in which B has to hold. You could add extra variables to your code to specify such states. This variable could be a JML ghost variable so the source code remains unchanged. When one of these boolean indicator variables is true the corresponding invariant has to hold in this state.

Alternatively you can have some JML command where you label some states over which you can reason about in KeY. In this example B is the invariant *m.res + m.i * a = 3 * a* which evaluates to true after each iteration of the loop. This example also makes use of the symbolic execution feature of KeY. The integer input variable `a` is not further specified, so the proofs holds for all possible values for `a`. In this example the same could be expressed by a loop invariant. However this approach is more general.

```
public class Multiply {
  public int res;
  public int i;

  public int times3(int a) {
    i = 3;
    res = 0;
    while(i > 0) {
      res = res + a;
      i = i - 1;
    }
    return res;
  }
}
```

Listing 18: Multiply.java

```
\javaSource "source/";

\programVariables {
  Multiply m;
  int a;
}

\problem {
  m != null ->
  {m.i := 0}{m.res := 0}
  \[[ {
    m.times3(a);
  }
  \]] (\box((\exists int u; (m.i = u & \wnext(m.i = u -
     1))) -> (m.res + m.i * a = 3 * a)))
}
```

Listing 19: times3.key

This proof can be completed totally automatically if *Loop treatment* is set to *expand*. The resulting proof tree is pictured in figure 9 . KeY automatically split the sequent to handle the case of a null reference that may be caused by *m*.

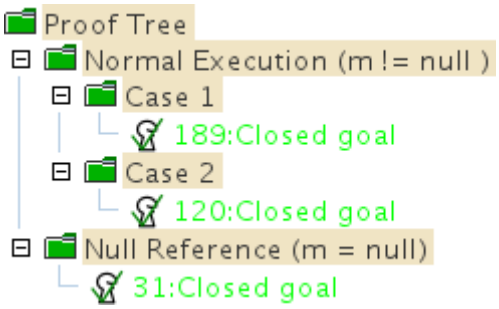


Figure 9: Proof tree of the Multiply example

9.6 Specifying a Bank Transaction

The objective of is example is to demonstrate how you can precisely specify with the help of trace operators when certain events happen during program execution. Often it is important to specify the order in which events happen. The bank example is a prototype of transactional behavior that can be described with the help of dynamic logic.

In *temporaljmlc* [HL10] temporal operators can be specified that refer to special events at the beginning and the end of method. In JavaDL with trace semantics however every assignment to an object field is considered a state transition in the trace. In this example that deals with a bank transfer, properties about the program are formulated that hold throughout the run. Specifying the behavior during a method and not just the result after exiting is important in cases where other processes can access the data during the method run or in cases where there is abnormal termination.

There are two classes. One called `Bank` and one called `Account`. The `Bank` class has a method `transfer` that transfers an amount of `c` from an account `acc1` to another account `acc2`. I exactly specify how the transaction should take place as follows:

At first the amount `c` first has to be withdrawn from `acc1`. After that the money has to be added the balance of `acc2`. From the way this is specified follows that the field variables `acc1` and `acc2` that contains the balance of each of the bank accounts only changes once. In the alternative scenario, called `Bank2`, the process of the transaction is slightly relaxed. Here we only demand that total amount of money is smaller or equal to the amount at the beginning throughout the whole transfer process. At the end the amount `c` has to be complete transferred from `acc1` to `acc2` in both cases. The proof finished automatically and has almost 1000 proof steps. There are many branching rules so the proof tree is rather big.

10 Outlook and Conclusion

There is the possibility to add further constructs to the JML extension. When specifying temporal properties one often wants to compare the value in some state to the value in the previous or the next state. This can achieved by using the $\exists$ quantifier as shown in section 9.2 and 9.3 . One could improve the usability by introducing the operators `\next` and `\prev` that are automatically translated into formulae using quantifiers.

In the current implementation of *KeY* with trace logic it is not possible to save proofs in which loop invariants occur. However that could be implemented. The bigger conceptual problem is to find a way to input the invariants not interactively but from an input file, preferably in the JML specification. The user of the verification tool still has to come up with suitable variants and invariants which is often complicated but the definition in a file then allows to repeat the verification automatically. In the work to this thesis the goal was to come up with a way to specify loop invariants for trace formula analogly to regular KeY. However the problem is that the same loop may need different instantiations of invariant and variant during the proof process as identified with the help of the example from section 9.4. In order to get along with just

one set of variant and invariant they would need to contain a complete trace themselves. This is currently not supported in the prototypical implementation an even then the proof process would be complicated for even this small example. The conclusion is that it is unlikely that larger specification problems can be solved with that approach.

The goal for the verifcation of trace properties cannot be to be able to prove all possible formulae. In general it is even undecidable to decide if a DTL or javaDTL is valid because of the undecidability of the halting problem. Because of that it would be a success to be able to prove some trace formula of practical relevance. The examples presented in this paper in section 9 demonstrate but it has to be possible to apply this process to bigger or more complicated instances for the use in practice. The example of the bank transaction in section 9.6 can be proved without user interaction. This example has just two assignments, needs 815 rule applications and about 2 seconds. Verification task of that sort could be extended a bit further. However with loops, even with finite ones, the correct decision whether to unroll the temporal operator or go on with symbolic execution has to bee made. Otherwise KeY ends up in a dead end where no rule application is possible any more. With these observations made in the work for this thesis the usefulness of KeY with javaDTL remains restricted. Further progress on the field of heuristics for rule selection have to be made to reduce user interaction. Only then the trace based verification approach can be successful.

```
\javaSource "source/";
\programVariables {
  Account acc1, acc2;
  Bank bank;
  int a, b, c;
  int c;
}

\problem {
  acc1 != null & acc2 != null & bank != null &
  acc1 != acc2 & c >= 0 ->
  {acc1.balance := a} {acc2.balance := b}
  \[[ {
    bank.transfer(acc1, acc2, c);
  }
  \]] ((acc1.balance = a) \until (\box acc1.balance = a -
      c) &
       (acc2.balance = b) \until (\box acc2.balance = b +
           c) &
       \box(acc1.balance + acc2.balance <= a + b))
}
```

Listing 22: Bank.Key

```
public class Bank {
    public void transfer(Account sender, Account receiver
        , int amount) {
    sender.balance = sender.balance - amount;
    receiver.balance = receiver.balance + amount;
    }
}
```

Listing 20: Bank.java

```
public class Account {
  public int balance;
}
```

Listing 21: Account.java

```
\javaSource "source/";
\programVariables {
  Account acc1, acc2;
  Bank bank;
  int a. b, c;
  int c;
}

\problem {
  acc1 != null & acc2 != null & bank != null &
  acc1 != acc2 & c >= 0 ->
  {acc1.balance := a} {acc2.balance := b}
  \[[ {
    bank.transfer(acc1, acc2, c);
  }
  \]] ((acc1.balance = a) \until (\box acc1.balance = a -
      c) &
       (acc2.balance = b) \until (\box acc2.balance = b +
           c) &
       \box(acc1.balance + acc2.balance <= a + b))
}
```

Listing 23: Bank2.key

References

[BB13] Bernhard Beckert and Daniel Bruns. Dynamic logic with trace semantics. In Maria Paola Bonacina, editor, *24th International Conference on Automated Deduction (CADE-24)*, number 7898 in Lecture Notes in Computer Science, pages 315–329. Springer-Verlag, 2013.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of java card programs. In *Revised Papers from the First International Workshop on Java on Smart Cards: Programming and Security*, JavaCard '00, pages 6–24, London, UK, UK, 2001. Springer-Verlag.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[HL10] Faraz Hussain and Gary T. Leavens. temporaljmlc: A jml runtime assertion checker extension for specification and checking of temporal properties. Technical Report CS-TR-10-08, UCF, Dept. of EECS, 2010.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hol03] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[LPC+08] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. JML Reference Manual. June 2008.

[TH02] Kerry Trentelman and Marieke Huisman. Extending jml specifications with temporal logic. In *Algebraic Methodology And Software Technology (AMAST'02), number 2422 in LNCS*, pages 334–348. Springer, 2002.

[Wei11] Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2011.