Proving Programs Incorrect using a Sequent Calculus for Java Dynamic Logic

Philipp Rümmer[1] and Muhammad Ali Shah[2]

[1] Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University
philipp@cs.chalmers.se
[2] Avanza Solutions ME, Dubai - 113116, United Arab Emirates
muhammad.ali@avanzasolutions.com

Abstract. Program verification is concerned with proving that a program is correct and adheres to a given specification. Testing a program, in contrast, means to search for a witness that the program is incorrect. In the present paper, we use a program logic for Java to prove the *incorrectness* of programs. We show that this approach, carried out in a sequent calculus for dynamic logic, creates a connection between calculi and proof procedures for program verification and test data generation procedures. Starting with a program logic enables to find more general and more complicated counterexamples for the correctness of programs.

Key words: Disproving, Program logics, Program verification, Testing

1 Introduction

Testing and program verification are techniques to ensure that programs behave correctly. The two approaches start with complementary assumptions: when we try to verify correctness, we implicitly expect that a program *is* correct and want to confirm this by conducting a proof. Testing, in contrast, expects incorrectness and searches for a witness (or *counterexample* for correctness):

"Find program inputs for which something bad happens."

In the present paper, we want to reformulate this endeavour and instead write it as an existentially quantified statement:

"There are program inputs for which something bad happens." (1)

Written like this, it becomes apparent that we can see testing as a proof procedure that attempts to eliminate the quantifier in statements of form (1). When considering functional properties, many program logics that are used for verification are general enough to formalise (1), which entails that calculi for such program logics can in fact be identified as testing procedures.

The present paper discusses how the statement (1), talking about a Java program and a formal specification of safety-properties, can be formalised in dynamic logic for Java [1, 2]. Through the usage of algebraic datatypes, this formalisation can be carried out without leaving first-order dynamic logic. Subsequently, we use a sequent calculus for automatic reasoning about the resulting formulae. The component of the calculus that is most essential in this setting is quantifier elimination. Depending on the way in which existential quantifiers are eliminated—by substituting ground terms, or by using metavariable techniques—we either obtain proof procedures that much resemble automated white-box test generation methods, or we arrive at procedures that can find more general and more complicated solutions (program inputs) of (1), but that are less efficient for "obvious" bugs. We believe that this viewpoint to incorrectness proofs can both lead to a better understanding of testing and to more powerful methods for showing that programs are incorrect.

Organisation of the Paper Sect. 2 introduces dynamic logic for Java and describes how (1) can be formalised. In Sect. 3, we show how different versions of a sequent calculus for dynamic logic can be used to reason about (1). Sect. 4 discusses how solutions of (1) can be represented. Sect. 5 provides further details about incorrectness proofs using the incremental closure approach. Sect. 6 discusses related work, and Sect. 7 gives future work and concludes the paper.

Running Example: Erroneous List Implementation The Java program shown in Fig. 1 is used as example in the whole paper. It is interesting for our purposes because it operates on a heap datastructure and contains unbounded loops, although it is not difficult to spot the bug in the method `delete`.

2 Formalisation of the Problem in Dynamic Logic

In the scope of this paper, the only "bad things" that we want to detect are violated post-conditions of programs. Arbitrary broken safety-properties (like assertions) can be reduced to this problem, whereas the violation of liveness-properties (like looping programs) falls in a different class and the techniques presented here are not directly applicable. This section describes how the statement that we want to prove can be formulated in dynamic logic:

> There is a pre-state—possibly subject to pre-conditions—such that the program at hand violates given post-conditions. (2)

Dynamic Logic First-order dynamic logic (DL) [1] is a multi-modal extension of first-order predicate logic in which modal operators are labelled with programs. There are primarily two kinds of modal operators that are dual to each other: a diamond formula $\langle\alpha\rangle\,\phi$ expresses that ϕ holds in at least one final state of program α. Box formulae can be regarded as abbreviations $[\alpha]\,\phi \equiv \neg\langle\alpha\rangle\,\neg\phi$ as usual. The DL formulae that probably appear most often have the form $\phi \rightarrow \langle\alpha\rangle\,\psi$ and state, for a deterministic program α, the total correctness of α concerning a precondition ϕ and a postcondition ψ. In this paper, we will only use dynamic logic for Java [2] (JavaDL) and assume that α is a list of Java statements.

```
public class IntList {
  private ListNode head;
  public void add (int n) { ... }

  /*@
    @ public normal_behavior
    @ ensures !contains(n);
    @*/
  public void delete(int n) {
    ListNode cur = head, prev = head;
    while (cur != null) {
      if (cur.val == n) prev.next = cur.next;
      else              prev = cur;
      cur = cur.next;
    }
  }

  public /*@ pure @*/ boolean contains(int n) {
    ListNode temp = head;
    while (temp != null) {
      if (temp.val == n) return true;
      temp = temp.next;
    }
    return false;
  }
}
```

```
class ListNode {
  public int      val;
  public ListNode next;
}
```

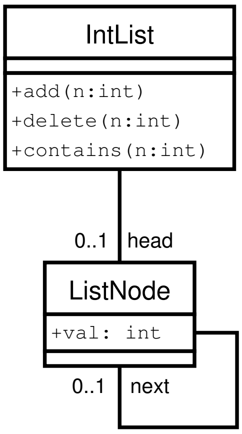

Fig. 1. The running example, a simple implementation of singly-linked lists, annotated with JML [3] constraints. We concentrate on the method `delete` for removing all elements with a certain value, which contains bugs.

Updates JavaDL features a notation for updating functions in a substitution-like style [4], which is primarily useful because it allows for a simple and natural memory representation during symbolic execution. For our purposes, *updates* can be seen as a simplistic programming language and are defined by the grammar:

$$\mathit{Upd} \ ::= \ \texttt{skip} \ \Big| \ f(s_1, \ldots, s_n) := t \ \Big| \ \mathit{Upd} | \mathit{Upd} \ \Big| \ \texttt{if} \ \phi \ \{\mathit{Upd}\} \ \Big| \ \texttt{for} \ x \ \{\mathit{Upd}\}$$

in which $s_1, \ldots, s_n, t$ range over terms, f over function symbols, ϕ over formulae and x over variables. The update constructors denote effect-less updates, assignments, parallel composition, guarded updates and quantified updates. Updates u can be attached to terms and formulae (like in $\{u\}\, t$) for changing the state in which the expression is supposed to be evaluated:

Expression with update:	Equivalent update-free expr.:
$\{a := g(3)\}\, f(a)$	f(g(3))
$\{x := y \mid y := x + 1\}\, (x < y)$	$y < x + 1$
$\{a := 3 \mid \texttt{for}\ x\ \{f(x) := 2 \cdot x + 1\}\}\, f(f(a))$	15

As illustrated here, it is always possible to apply updates to terms and formulae like a substitution, unless a formula contains further modal operators. In the latter case, the application has to be delayed until the modal operator is eliminated.

2.1 Heap Representation in Dynamic Logic for Java

Reasoning in JavaDL always takes place in the context of a system of Java classes, which is supposed to be free of compile-time errors. From this context, a vocabulary of sorts and function symbols is derived that represents variables and the heap of the program in question [2].

Most importantly, in JavaDL objects of classes are identified with natural numbers. For each class C, a sort with the same name and a (injective) function $C.get : nat \to C$ are introduced. $C.get(i)$ is the ith object of class C (i is the index or "address"). For distinct classes C and D, $C.get(i)$ and $D.get(j)$ are never the same object. Each sort C representing a class also contains a distinguished individual denoted by *null*, which is used to represent undefined references. Attributes of type T of a class C are modelled by functions $C \to T$. Instead of the infix notation $attr(o)$, we mostly write $o.attr$ for attribute accesses.

C can be seen as a reservoir containing both those objects that are already created and those that can possibly be created later by a program: JavaDL uses a constant-domain semantics in which modal operators never change the domains of existing individuals. In order to distinguish existing and non-existing objects, for each class C also a constant $C.nextToCreate : nat$ is declared that denotes the lowest index of a non-created object. All objects $C.get(i)$ with $i < C.nextToCreate$ are created, all others are not.

For the program in Fig. 1, the vocabulary is as follows:

Sorts:	Functions:		
IntList, *ListNode*,	*IntList.get*	:	$nat \to IntList$
int, *nat*, ...	*ListNode.get*	:	$nat \to ListNode$
	IntList.nextToCreate	:	*nat*
	ListNode.nextToCreate	:	*nat*
	head	:	$IntList \to ListNode$
	next	:	$ListNode \to ListNode$
	val	:	$ListNode \to int$

2.2 Formalising the Violation of Post-Conditions

We go back to (2). It is almost straightforward to formalise the part of (2) that comes after the existential quantifier "there is a pre-state":

$$\neg\big(\textit{pre-conditions} \to \langle\, \textit{statements} \,\rangle\ \textit{post-conditions}\big) \tag{3}$$

Formula (3) is true if and only if the pre-conditions hold, the program fragment does not terminate, or terminates and the post-conditions do not hold in the final state.

Property (2) does not mention termination, which could be interpreted in different ways. If in (3) the box operator $[\alpha]\,\phi$ was used instead of a diamond, we would also specify that the program has to terminate for the inputs that we search for. JavaDL does, however, not distinguish between non-termination due to looping and abrupt termination due to exceptions (partial correctness model). Because we, most likely, will consider abrupt termination as a violation of the post-conditions, the diamond operator appears more appropriate.

2.3 Quantification over Program States

In order to continue formalising (2), it is necessary to close the statement (3) existentially and to add quantifiers that express "there is a pre-state":

$$\exists\, \mathit{pre\text{-}state}.\ \{\mathit{pre\text{-}state}\}\,\neg\big(\mathit{pre\text{-}conditions} \rightarrow \langle\, \mathit{statements}\,\rangle\ \mathit{post\text{-}conditions}\big) \qquad (4)$$

Because state quantification is not directly possible in JavaDL, we use an update $\{\mathit{pre\text{-}state}\}$ to define the state in which (3) is to be evaluated. For a Java program, the pre-state covers (i) variables that turn up in a program, and (ii) the heap that the program operates on. Following Sect. 2.1, at a first glance this turns out to be a second-order problem, because the heap is modelled by functions like *head*, *next*, etc.[3] A second glance reveals, fortunately, that a proper Java program and proper pre- and post-conditions[4] will only look at the values $C.\mathit{get}(i).\mathit{attr}$ of attributes for $i < C.\mathit{nextToCreate}$: the state of non-existing objects is irrelevant. Quantification of $C.\mathit{nextToCreate}$ and the finite prefix

$$C.\mathit{get}(0).\mathit{attr},\ C.\mathit{get}(1).\mathit{attr},\ \ldots,\ C.\mathit{get}(C.\mathit{nextToCreate}-1).\mathit{attr}$$

can naturally be realised through quantification over algebraic datatypes like lists. Note, that the number of quantified locations is finite, but unbounded.

Attributes of Primitive Types The simplest case is an attribute *attr* of a primitive Java type. If *attr* has type *int*, the quantification can be performed as follows:

$$\exists\, \mathit{attr}_V : \mathit{intList}.\ \ \{\texttt{for}\ x : \mathit{nat}\ \{C.\mathit{get}(x).\mathit{attr} := \mathit{attr}_V \downarrow x\}\}\ \ldots$$

Apart from the actual quantifier, an update is used for copying the contents of the list variable attr_V to the attribute. The expression also contains an operator for accessing lists $[a_0, \ldots, a_n]$, which we define by

$$[a_0, \ldots, a_n] \downarrow i\ :=\ \begin{cases} a_i & \text{for } i \leq n \\ 0 & \text{otherwise} \end{cases} \qquad (i : \mathit{nat})$$

The fact that the operator returns a default value (0, but any other value would work equally well) for accesses outside of the list bounds simplifies the overall treatment and basically renders the length of lists irrelevant. Instead of lists, one could also talk about functions with finite support.

[3] JavaDL does not provide higher-order quantification.

[4] In the whole paper, we assume that pre- and post-conditions only talk about the program state, and only about created objects.

Attributes of Reference Types The quantification is a bit more involved for attributes *attr* of type D, where D is a reference type like a class: (i) attributes can be undefined, i.e., have value *null*, (ii) attributes of created objects must not point to non-created objects, and (iii) attributes of type D can also point to objects of type D', provided that D' is a subtype of D. We capture these requirements by overloading the function $D.get$. Assuming that $D_0\,(= D), \ldots, D_k$ is an arbitrary, but fixed enumeration of D's subtypes, we define:

$$D.get(s,i) \;:=\; \begin{cases} D_s.get(i) & \text{for } i < D_s.nextToCreate,\ s \le k \\ null & \text{otherwise} \end{cases} \qquad\qquad (s,i : nat)$$

Apart from the object index i, we also pass $D.get(s,i)$ the index s of the requested subtype of D. The result of $D.get(s,i)$ is either a created object (if i and s are within their bounds $D_s.nextToCreate$ and k) or *null*. With this definition, the quantification part for a reference attribute boils down to

$$\exists\, a_S, a_V : natList.\ \ \{\texttt{for}\ x : nat\ \{C.get(x).attr := D.get(a_S \downarrow x, a_V \downarrow x)\}\}\ \ldots$$

In case of a class D that does not have proper subclasses, the list a_S can of course be left out (and the first argument of $D.get$ can be set to 0).

Example We show the formalisation of (2) for the method `delete` in the program of Fig. 1. Apart from the values of the attributes *head*, *next* and *val*, which are treated as discussed above, one also has to quantify over the number of created objects (*IntList.nextToCreate* and *ListNode.nextToCreate*), over the receiver o of the method invocation and over the argument n. o is assumed to be either an arbitrary created object or *null* ($IntList.get(0, o_V)$). The pre- and post-conditions correspond to the JML specification: initially, o is not *null*, and `delete` in fact removes the elements with value n.

$$\begin{array}{l}
\exists\, k_{IL}, k_{LN}, o_V : nat.\ \ \exists\, n_V : int.\ \ \exists\, head_V, next_V : natList.\ \ \exists\, val_V : intList.\\
\ \ \{IntList.nextToCreate := k_{IL} \mid ListNode.nextToCreate := k_{LN}\}\\
\ \ \{\texttt{for}\ x : nat\ \{IntList.get(x).head := ListNode.get(0, head_V \downarrow x)\} \mid\\
\ \ \ \ \texttt{for}\ x : nat\ \{ListNode.get(x).next := ListNode.get(0, next_V \downarrow x)\} \mid\\
\ \ \ \ \texttt{for}\ x : nat\ \{ListNode.get(x).val := val_V \downarrow x\} \mid\\
\ \ \ \ o := IntList.get(0, o_V) \mid n := n_V\}\\
\ \ \ \ \ \ \neg\big(\ o \neq null \rightarrow \langle\, o.\texttt{delete}(n)\,\rangle\,\langle\, b = o.\texttt{contains}(n)\,\rangle\, b = FALSE\ \big)
\end{array} \qquad (5)$$

3 Constructing Proofs for Program Incorrectness

A Gentzen-style sequent calculus for JavaDL is introduced in [2], which has been implemented in the KeY system and is used by us as test-bed. Fig. 2 shows a small selection of the rules. Relevant for us are the following groups of rules: (i) rules for a sequent calculus for first-order predicate logic with metavariables (the first 5 rules of Fig. 2), (ii) rules that implement symbolic execution [5] for Java (the

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}\ \wedge\textsc{r} \qquad \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \wedge\textsc{l} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \neg\textsc{r}$$

$$\frac{\Gamma \vdash \phi[x/f(X_1, \ldots, X_n)], \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\ \forall\textsc{r} \qquad (X_1, \ldots, X_n \text{ all metavariables in } \phi)$$

$$\frac{\Gamma \vdash \phi[x/X], \exists x.\phi, \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\ \exists\textsc{r} \qquad (X \text{ a fresh metavariable})$$

$$\frac{\Gamma, \{u\}\, \{r := l\}\, \langle \ldots \rangle\, \phi \vdash \Delta}{\Gamma, \{u\}\, \langle r = l; \ldots \rangle\, \phi \vdash \Delta}\ \textsc{assign-l} \qquad (r, l \text{ side-effect-free})$$

$$\frac{\begin{array}{c}\Gamma, \{u\}\, \langle \alpha_1; \ldots \rangle\, \phi, \{u\}\, b \vdash \Delta \\ \Gamma, \{u\}\, \langle \alpha_2; \ldots \rangle\, \phi \vdash \{u\}\, b, \Delta\end{array}}{\Gamma, \{u\}\, \langle \texttt{if}\ (b)\ \alpha_1\ \texttt{else}\ \alpha_2\ \ldots \rangle\, \phi \vdash \Delta}\ \textsc{if-l} \qquad (b \text{ side-effect-free})$$

$$\frac{\Gamma, \{u\}\, \langle \texttt{if}\ (b)\ \{\alpha;\ \texttt{while}\ (b)\ \alpha\} \ldots \rangle\, \phi \vdash \Delta}{\Gamma, \{u\}\, \langle \texttt{while}\ (b)\ \alpha\ \ldots \rangle\, \phi \vdash \Delta}\ \textsc{while-l}$$

Fig. 2. Examples of (simplified) sequent calculus rules for JavaDL. In the last three rules, the update u can also be empty (`skip`) and disappear. Γ and Δ denote arbitrary sets of formulae (side-formulae).

last three rules of Fig. 2), and (iii) rewriting rules for applying and simplifying updates (not shown here, see [4]). The rule ASSIGN-L turns a Java assignment into an update, which subsequently can be merged with the former preceding update u and simplified. In IF-L, a case analysis for an if-statement is performed by splitting on the branch predicate b evaluated in the current program state u. Both rules require that expressions with side-effects are simplified first. Finally, the rule WHILE-L unwinds a loop once.

The fact that the calculus directly integrates symbolic execution—and covers all important features of Java like dynamic object creation and exceptions—is most central for us. When symbolically executing a program, the proof tree resembles the *symbolic execution tree* of the program [5] and reflects the (feasible) paths through the program. Branch predicates that describe, in terms of the pre-state, when a certain path is taken are accumulated as formulae in a sequent. JavaDL introduces such predicates for conditional statements and for statements that might raise exceptions. A simple example is the following proof:

$$\frac{\dfrac{\dfrac{\vdots}{p + 1 \leq 0, p \geq 0 \vdash}}{\dfrac{\{p := p + 1\}\, \langle\rangle\, p \leq 0, p \geq 0 \vdash}{\langle p = p + 1; \rangle\, p \leq 0, p \geq 0 \vdash}}\ \textsc{assign-l} \qquad \dfrac{\dfrac{\dfrac{\vdots}{-p \leq 0 \vdash p \geq 0}}{\{p := -p\}\, \langle\rangle\, p \leq 0 \vdash p \geq 0}}{\langle p = -p; \rangle\, p \leq 0 \vdash p \geq 0}}{\langle \texttt{if}\ (p \geq 0)\ p = p + 1;\ \texttt{else}\ p = -p; \rangle\, p \leq 0 \vdash}\ \textsc{if-l}$$

Symbolic execution and update application can usually be automated easily—in contrast to reasoning in first-order logic—because in each proof situation only few rules are applicable, and because the application order does not matter.

This section discusses how the sequent calculus can be used to prove formulae (4). The first and essential task is always to eliminate the existential quantifiers, i.e., to provide the programs inputs, which can be concrete or symbolic. Assuming that pre- and post-conditions only talk about the program state, it is sufficient to apply $\exists\text{R}$ once (and not multiple times) for each quantifier in $\exists\, \textit{pre-state}$, because the validity of (4) only depends on the program fragment and the pre- and post-conditions, not on the values of other symbols.

We focus on and propose two methods for constructing proofs: the usage of metavariables and depth-first search (Sect. 3.2) and the usage of metavariables and backtracking-free search with constraints (Sect. 3.3, Sect. 5). In our experiments, we have concentrated on the latter method, because the implementation KeY follows this paradigm. As a comparison, Sect. 3.1 shortly discusses how a ground calculus would handle (4), which resembles common test generation techniques.

3.1 Construction of Proofs using a Ground Proof Procedure

The simplest approach is *ground reasoning*, i.e., to not use metavariables. Therefore, a ground version of $\exists\text{R}$ can be used: (t is an arbitrary term)

$$\frac{\Gamma \;\vdash\; \phi[x/t], \exists x.\phi, \Delta}{\Gamma \;\vdash\; \exists x.\phi, \Delta}\ \exists\text{R}_g$$

Equivalently, also the normal rule $\exists\text{R}$ can be applied, immediately followed by a *substitution step* that replaces the introduced metavariable X with a concrete term t. For (4), the usage of rule $\exists\text{R}_g$ encompasses that a concrete pre-state has to be chosen up-front that satisfies the pre-condition and makes the program violate its post-condition. If we consider (5), for instance, we see that a proof can be conducted with the following instantiations:

k_{IL}	k_{LN}	o_V	n_V	head_V	next_V	val_V
1	1	0	5	[0]	[7]	[5]

(6)

The instantiations express that the classes *IntList* and *ListNode* have one created object each (k_{IL}, k_{LN}), that the object $\mathit{IntList.get}(0)$ receives the method invocation (o_V) with argument 5 (n_V), that $\mathit{IntList.get}(0).\mathit{head}$ points to the object $\mathit{ListNode.get}(0)$ (head_V), that $\mathit{ListNode.get}(0).\mathit{next}$ is *null* (next_V, because of $7 \geq k_{LN}$), i.e., that the receiving list has only one element, and that $\mathit{ListNode.get}(0).\mathit{val}$ is 5 (val_V).

A ground proof of a formula (4) is the most specific description of an erroneous situation that is possible. For debugging purposes, this is both an advantage and a disadvantage: (i) it is possible to concretely follow a program execution that leads to a failure, but (ii) the description does not distinguish

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}*\\ {[\,P \mapsto 2\,]}\end{array}}{P+1>3, P \geq 0 \ \vdash}}{\{p := P+1\}\,\langle\rangle\, p>3, p \geq 0 \ \vdash}}{\{p := P\}\,\langle p = p+1;\rangle\, p>3, P \geq 0 \ \vdash} \qquad \dfrac{\begin{array}{c}*\\ {[\,P \mapsto 2\,]}\end{array}}{\{p := P\}\,\langle p = -p;\rangle\, p>3 \ \vdash \ P \geq 0}}{\{p := P\}\,\langle \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p;\rangle\, p>3 \ \vdash}\ \text{IF-L}}{\vdash \ \neg\{p := P\}\,\langle \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p;\rangle\, p>3,\ \ldots}\ \neg\text{R}}{\vdash \ \exists p_V : \mathit{int}.\ \{p := p_V\}\,\neg\langle \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p;\rangle\, p>3}\ \exists\text{R}
$$

Fig. 3. Proof that a program violates its post-condition $p > 3$. The initial (quantified) formula is derived as described in Sect. 2. The application of updates is not explicitly shown in the proof.

between those inputs (or input features) that are relevant for causing a failure and those that are irrelevant. The disadvantage can partly be undone by looking at more than one ground proof, and by searching for proofs with "minimal" input data (e.g., [6]). Technically, the main advantage of a ground proof is that program execution (and checking pre- and post-conditions) is most efficient for a concrete pre-state. The difficulty, of course, is to find the *right* pre-state, which is subject of techniques for automated test data generation. Common approaches are the generation of *random* pre-states (e.g., [6]), or the usage of backtracking, symbolic execution and constraint techniques in order to optimise coverage criteria and to reach the erroneous parts of a program (see, e.g., [7]).

3.2 Construction of Proofs using Metavariables and Backtracking

The most common technique for efficient automated proof search in tableau or sequent calculi are rigid metavariables (also called free variables) and backtracking (depth-first search), for an overview see [8]. The rules shown in Fig. 2, together with a global substitution rule that allows to substitute terms for metavariables in a proof tree, implement a corresponding sequent calculus. Because, in particular, the substitution rule is destructive and a wrong decision can hinder the subsequent proof construction, proof procedures usually carry out a depth-first search with iterative deepening and backtrack when earlier rule applications appear misleading.

The search space of a proof procedure can be seen as an and/or search tree: (i) And-nodes occur when the proof branches, for instance when applying $\wedge$R, because each of the new proof goals has to be closed at some point. (ii) Or-nodes occur when a decision has to be drawn about which rule to apply next, or about a substitution that should be applied to a proof; in general, only one of the possible steps can be taken.

Metavariables and backtracking can be used to prove formulae like (4). The central difference to the ground approach is that metavariables can be introduced as place-holders for the pre-state, which can later be refined and made concrete by applying substitutions. A simple example is shown in Fig. 3, where the initial value of the variable p is represented by a metavariable P. After symbolic execution of the program, it becomes apparent that the post-condition $p > 3$ can be violated in the left branch by substituting 2 for P. The right branch can then be closed immediately, because this path of the program is not executed for $P = 2$: the branch predicate $P \geq 0$ allows to close the branch. Generally, the composition of the substitutions that are applied to the proof can be seen as a description of the pre-state that is searched for. A major difference to the ground case is that a substitution also can describe *classes* of pre-states, because it is not necessary that concrete terms are substituted for all metavariables.

Branch Predicates Strictly speaking, the proof branching that is caused by the rule IF-L (or by similar rules for symbolic execution) falls into the "and-node" category: all paths through the program have to be treated in the proof. The situation differs, however, from the branches introduced by $\wedge$R, because IF-L performs a cut (a case distinction) on the branch predicate $\{u\}\, b$. As the program is executed with symbolic inputs (metavariables), it is possible to turn $\{u\}\, b$ into *true* or *false* (possibly into both, as one pleases), by applying substitutions and choosing the pre-state appropriately. Coercing $\{u\}\, b$ in this way will immediately close one of the two branches.

There are, consequently, two principal ways to close (each of) the proof branches after executing a conditional statement: (i) the program execution can be continued until termination, and the pre-state can be chosen so that the post-condition is violated, or (ii) one of the two branches can be closed by making the branch predicate *true* or *false*, which means that the program execution is simply forced *not* to take the represented path. Both cases can be seen in Fig. 3, in which the same substitution $P \mapsto 2$ leads to a violation of the post-condition in the left branch and turns the branch predicate in the right branch into *true*.

Proof Strategy The proof construction consists of three parts: (i) pre-conditions have to be proven, (ii) the program has to be executed symbolically in order to find violations of the post-conditions, and (iii) it has to be ensured that the program execution takes the right path by closing the remaining proof branches with the help of branch predicates. These steps can be performed in different orders, or also interleaved. Furthermore, it can in all phases be necessary to backtrack, for instance when a violation of the post-conditions was found but the pre-state does not satisfy the pre-condition, or if the path leading to the failure is not feasible.

Example Formula (5) can be proven by choosing the following values, which could be found using metavariables and backtracking:

k_{IL}	k_{LN}	o_V	n_V	$head_V$	$next_V$	val_V
1	1	0	N_V	$[0, \ldots]$	$[7, \ldots]$	$[N_V, \ldots]$

(7)

Comparing this solution to (6), the main difference is that no concrete value has to be chosen for n_V. It suffices to state that the value of n_V coincides with the first element of the list val_V: when calling `delete`, the actual parameter coincides with the first element of the receiving linked list. Likewise, the parts of the pre-state that are described by lists do not have to be determined completely: the tail of lists can be left unspecified by applying substitutions like $VAL_V \mapsto \mathrm{cons}(N_V, VAL_{tail})$ (which is written as $[N_V, \ldots]$ in the table). Sect. 4 discusses how the representation of solutions can further be generalised.

3.3 Construction of Proofs using Incremental Closure

There are alternatives to proof search based on backtracking: one idea is to work with metavariables, but to delay the actual application of substitutions to the proof tree until a substitution has been found that closes all branches. The idea is described in [9] and worked out in detail in [10]. While backtracking-free proof search is, in principle, also possible when immediately applying substitutions, removing this destructive operation vastly simplifies proving without backtracking. Because KeY implements this technique, it is used in our experiments.

The approach of [10] works by explicitly enumerating and collecting, for each of proof goals, the substitutions that would allow to close the branch. Substitutions are represented as *constraints*, which are conjunctions of unification conditions $t_1 \equiv t_2$. A generalisation is discussed in Sect. 4. For the example in Fig. 3, the "solutions" of the left branch could be enumerated as $[\,P \equiv 2\,]$, $[\,P \equiv 1\,]$, $[\,P \equiv 0\,]$, $[\,P \equiv -1\,]$, ..., and the solutions of the right branch as $[\,P \equiv 0\,]$, $[\,P \equiv 1\,]$, $[\,P \equiv 2\,]$, ... In this case, we would observe that, for instance, the substitution represented by $[\,P \equiv 0\,]$ closes the whole proof. Generally, the conjunction of the constraints for the different branches describes the substitution that allows to close a proof (provided that it is consistent).

When proving formulae (4) using metavariables, a substitution (i.e., pre-state) has to be found that simultaneously satisfies the pre-conditions, violates the post-conditions in one (or multiple) proof branches and invalidates the branch predicates of all remaining proof branches. The constraint approach searches for such a substitution by enumerating the solutions of all three in a fair manner. In our experiments, we also used breadth-first exploration of the execution tree of programs, which simply corresponds to a fair selection of proof branches and formulae that rules are applied to. For formula (5), the method could find the same solution (7) as the backtracking approach of Sect. 3.2.

Advantages Compared to backtracking, the main benefits of the constraint approach are that duplicated rule applications (due to removed parts of the proof tree that might have to be re-constructed) are avoided, and that it is possible to search for different solutions in parallel. Because large parts of the proofs in question—the parts that involve symbolic execution—can be constructed algorithmically and do not require search, the first point is particularly significant here. The second point holds because the proof search does never commit to one particular (partial) solution by applying a substitution. Constraints also naturally lead to more powerful representations of classes of pre-states (Sect. 4).

Disadvantages Destructively applying substitutions has the effect of *propagating* decisions that are made in one proof branch to the whole proof. While this is obviously a bad strategy for wrong decisions, it is by far more efficient to *verify* a substitution that leads to a solution (by applying it to the whole proof and by closing the remaining proof branches) than to hope that the remaining branches can independently come up with a compatible constraint. In Fig. 3, after applying the substitution $[\,P \mapsto 2\,]$ that is found in the left branch, the only work left in the right branch is to identify the inequation $2 \geq 0$ as valid. Finding a common solution of $P + 1 \not> 3$ and $P \geq 0$ by enumerating partial solutions, in contrast, is more naive and less efficient. One aspect of this problem is that unification constraints are not a suitable representation of solutions when arithmetic is involved (Sect. 4).

3.4 A Hybrid Approach: Backtracking and Incremental Closure

Backtracking and non-destructive search using constraints do not exclude each other. The constraint approach can be seen as a more fine-grained method for generating substitution candidates: while the pure backtracking approach always looks at a single goal when deriving substitutions, constraints allow to compare the solutions that have been found for multiple goals. The number of goals that can simultaneously be closed by one substitution, for instance, can be considered as a measure for how reasonable the substitution is. Once a good substitution candidate has been identified, it can also be applied to the proof destructively and the proof search can continue focussing on this solution candidate. Because the substitution could, nevertheless, be misleading, backtracking might be necessary at a later point. Such hybrid proof strategies have not yet been developed or tested, to the best of our knowledge.

4 Representation of Solutions: Constraint Languages

In Sect. 3.2 and 3.3, classes of pre-states are represented as substitutions or unification constraints. These representations are well-suited for pure first-order problems [10], but they are not very appropriate for integers (or natural numbers) that are common in Java: (i) Syntactic unification does not treat interpreted functions like $+$, $-$ or literals in special way. This rules out too many constraints, for instance $[\,X + 1 \equiv 2\,]$, as inconsistent. (ii) Unification conditions $t_1 \equiv t_2$ cannot describe simple classes of solutions that occur frequently, for instance classes that can be described by linear conditions like $X \geq 0$.[5]

[5] Depending on the representation of integers or natural numbers, certain inequations like $X \geq 1 \Leftrightarrow X \equiv \mathrm{succ}(X')$ might be expressible, but this concept is rather restricted.

The constraint approach of Sect. 3.3 is not restricted to unification constraints: we can see constraints in a more semantic way and essentially use any sub-language of predicate logic (also in the presence of theories like arithmetic) that is closed under the connective $\wedge$ as constraint language. For practical purposes, validity should be decidable in the language, although this is not strictly necessary. The language that we started using in our experiments is a combination of unification conditions (seen as equations) and linear arithmetic:

$$C \;::=\; C \wedge C \;\big|\; t_{int} = t_{int} \;\big|\; t_{int} \neq t_{int} \;\big|\; t_{int} < t_{int} \;\big|\; t_{int} \leq t_{int} \;\big|\; t_{oth} = t_{oth}$$

in which t_{int} ranges over terms of type *int* and t_{oth} over terms of other types. The constraints are given the normal model-theoretic semantics of first-order formulae (see, for instance, [9]):

Definition 1. *A constraint C is called* consistent *if for each arithmetic structure (interpreting the symbols $+$, $-$, $\neq$, $<$, $\leq$ and literals as is common over the integers, and all other function symbols arbitrarily), there is an assignment of values to metavariables such that C is evaluated to tt.*

Example 1. Of the following constraints, C_1, C_2 and C_3 are consistent, while the others are not. C_4 is inconsistent because the ranges of f and g could be disjoint, C_5 because f could be the identity, and C_6 because 5 could be outside of the range of the function $\cdot\!\downarrow\!\cdot$. Our constraint language does not know about lists, so that $\cdot\!\downarrow\!\cdot$ is just an arbitrary function symbol in this regard.

$$\begin{array}{lcl@{\qquad}lcl}
C_1 & := & X = 5 \wedge 2 = Y + 1 & C_2 & := & h(A, 2) = h(h(c, Y), Y + 1) \\
C_3 & := & c < X \wedge d \leq X & C_4 & := & f(X) = g(Y) \\
C_5 & := & X < f(X) & C_6 & := & (ATTR\!\downarrow\!O) = 5
\end{array}$$

We are in the process of working out details of this language—so far, we do not know whether consistency of constraints is decidable. Using a prototypical implementation of the constraints in KeY (as part of the constraint approach of Sect. 3.3), it is possible to find the following solution of (5) automatically:

k_{IL}	k_{LN}	o_V	n_V	$head_V$	$next_V$	val_V
K_{IL}	K_{LN}	0	N_V	$[0, \ldots]$	$[E, \ldots]$	$[N_V, \ldots]$

$$\begin{array}{l} K_{IL} > 0 \;\wedge \\ K_{LN} > 0 \;\wedge \\ E \geq K_{LN} \end{array}$$

Compared to (7), this description of pre-states is more general and no longer contains the precise number of involved objects of *IntList* and *ListNode*. It is enough if at least one object of each class is created ($K_{IL} > 0$, $K_{LN} > 0$). Further, the solution states that *IntList.get*(0) receives the invocation of `delete` with arbitrary argument N_V, that *IntList.get*(0)*.head* points to the object *ListNode.get*(0), that the attribute *ListNode.get*(0)*.next* is *null* ($E \geq K_{LN}$), i.e., the receiving list has only one element, and that the value of this element coincides with N_V.

5 Reasoning about Lists and Arithmetic

The next pages give more (implementation) details and treat some further aspects of the backtracking-free method from Sect. 3.3. As incremental closure works by enumerating the closing constraints of all proof branches, the central issue is to design suitable goal-local rules that produce such constraints, and to develop an application strategy that defines which rule should be applied at which point in a proof. The solutions shown here are tailored to the constraint language of the previous section.

5.1 Rules for the Theory of Lists

For proof obligations of the form (4), the closing constraints of a goal mostly describe the values of metavariables X_1, X_2, … over lists—the lists that in Sect. 2.3 are used to represent program states—and usually have the form:

$$\begin{aligned} &X_1 = \mathit{cons}(X_1^1, \mathit{cons}(X_1^2, \ldots)) \ \wedge \ X_2 = \mathit{cons}(X_2^1, \mathit{cons}(X_2^2, \ldots)) \ \wedge \ \cdots \\ \wedge \ &C(X_1^1, X_1^2, \ldots, X_2^1, X_2^2, \ldots) \end{aligned}$$

Such constraints consist of a first part that determines to which depth the lists $X_1, X_2, \ldots$ have been "expanded," and of a part $C(X_1^1, X_1^2, \ldots, X_2^1, X_2^2, \ldots)$ (which is again a constraint, e.g. in the language from Sect. 4) that describes the values of list elements. As each of the list elements X_1^1, X_1^2, … belongs to one object of a class (following Sect. 2.3), this intuitively means that a constraint always represents one fixed arrangement of objects in the heap. One constraint in the language from Sect. 4 cannot represent multiple isomorphic heaps (like heaps that only differ in the order of objects), because the constraints are not evaluated modulo the theory of lists. As it is explained in Example 1, a constraint like $(\mathit{ATTR} \downarrow O) = 5$, telling that the value of an instance attribute is 5 for the object with index O, is inconsistent and has to be written in a more concrete form like $\mathit{ATTR} = \mathit{cons}(\mathit{ATTR}^1, T) \wedge O = 0 \wedge \mathit{ATTR}^1 = 5$.

The expansion of lists is handled by a single rule that introduces fresh metavariables H, T for the head and the tail of a list. We use the *constrained formula* approach from [10] to remember this decomposition of a list L into two parts. A constrained formula is a pair $\phi \ll C$ consisting of a formula ϕ and a constraint C. The semantics of a formula $\phi \ll C$ that occurs in the antecedent of a sequent is (roughly) the same as of the implication $C \rightarrow \phi$, and in the succedent the semantics is $C \wedge \phi$: intuitively, the presence of ϕ can only be assumed if the constraint C holds. C has to be kept and propagated to all formulae that are derived from $\phi \ll C$ during the course of a proof. If $\phi \ll C$ is used to close a proof branch, the closing constraint that is created has to be conjoined with C.

The rule for expanding lists is essentially a case distinction on whether the head ($i = 0$) or a later element ($i > 0$) of a list is accessed. An attached constraint $[\, L = \mathit{cons}(H, T) \,]$ expresses that the name H is introduced for the head of the list and T for its tail. In practice, the rule is only applied if an expression $L \downarrow i$ occurs in the sequent $\Gamma \ \vdash \ \Delta$, where L is a metavariable. As described in Sect. 2.3, the

$$\begin{array}{c} * \\ [\,H > 3 \land C\,] \\ \hline X = 0 \ll C, L{\downarrow}0 = H \ll C, H \le 3 \ll C \;\vdash \\ \hline X = 0 \ll C, L{\downarrow}0 = H \ll C \;\vdash\; H > 3 \ll C \\ \hline X = 0 \ll C, L{\downarrow}0 = H \ll C \;\vdash\; L{\downarrow}X > 3 \\ \hline (X = 0 \land L{\downarrow}0 = H) \ll C \;\vdash\; L{\downarrow}X > 3 \\ \hline \mathcal{D} \end{array} \quad {\le}\mathrm{L}$$

$$\dfrac{\mathcal{D} \qquad \begin{array}{c} * \\ [\,X < 1 \land C\,] \\ \hline X \ge 1 \ll C, L{\downarrow}X = T{\downarrow}(X-1) \ll C \;\vdash\; L{\downarrow}X > 3 \\ \hline X > 0 \ll C, L{\downarrow}X = T{\downarrow}(X-1) \ll C \;\vdash\; L{\downarrow}X > 3 \\ \hline (X > 0 \land L{\downarrow}X = T{\downarrow}(X-1)) \ll C \;\vdash\; L{\downarrow}X > 3 \end{array} {\ge}\mathrm{L}}{\vdash\; L{\downarrow}X > 3}\;\textsc{expand-list}$$

Fig. 4. Example for a proof involving lists and metavariables $L, T : \mathit{intList}$, $H : \mathit{int}$, $X : \mathit{nat}$. We write C as abbreviation for the constraint $[\,L = \mathit{cons}(H,T)\,]$. The first solution (shown here) that is produced by the proof is $[\,L = \mathit{cons}(H,T) \land X < 1 \land H > 3\,]$ and stems from the formulas $X \ge 1 \ll C$ and $H \le 3 \ll C$ in the two branches. When applying further rules to the proof—instead of closing it—and expanding the list more than once, further solutions like $[\,L = \mathit{cons}(H, \mathit{cons}(H', T')) \land X = 1 \land H' > 3\,]$ can be generated. Concerning the handling of inequations in the proof, see Sect. 5.3.

length of lists is irrelevant, so that the case $L = \mathit{nil}$ does not have to be taken into account:

$$\dfrac{\begin{array}{c} \Gamma, (i = 0 \land (L{\downarrow}0) = H) \ll [\,L = \mathit{cons}(H,T)\,] \;\vdash\; \Delta \\ \Gamma, (i > 0 \land (L{\downarrow}i) = (T{\downarrow}(i-1))) \ll [\,L = \mathit{cons}(H,T)\,] \;\vdash\; \Delta \end{array}}{\Gamma \;\vdash\; \Delta}\;\textsc{expand-list}$$

$$(H, T \text{ fresh metavariables})$$

Fig. 4 shows an example how EXPAND-LIST is used to enumerate the solutions of the formula $L{\downarrow}X > 3$.

By repeated application of EXPAND-LIST, all list access expressions $L{\downarrow}i$ in a sequent can be replaced with scalar metavariables, which subsequently can be handled with other rules for first-order logic and arithmetic. The fact that different goals are created for all possible heap arrangements (because EXPAND-LIST splits on the value of the list index i) obviously leads to a combinatorial explosion, however, when the number of considered objects is increased. This is not yet relevant for programs like the one in Fig. 1. Generally, two possibilities to handle this issue (which we have not investigated yet) are (i) to work with a constraint language that directly supports the theory of lists, or to (ii) use the approach suggested in Sect. 3.4 to focus on one particular heap arrangement, ignoring isomorphic heaps. In this manner, it is, for instance, possible to simulate the lazy-initialisation approach from [11].

5.2 Fairness Conditions

As the different branches (and formulae) of a proof are expanded completely independently when using incremental closure, it is important to choose a fairness strategy that ensures an even distribution of rule applications. When proving program incorrectness, there are two primary parameters that describe how far a problem has been explored: (i) how often loops have been unwound on a branch (the number of applications of the rule WHILE-L from Fig. 2), and the (ii) the depth to which lists have been expanded (the size of the heap under consideration, or the number of applications of the rule EXPAND-LIST from the previous section).

In the KeY prover, automatic reasoning is controlled by *strategies*, which are basically cost computation functions that assign each possible rule application in a proof an integer number as cost. The rule application that has been given the least cost (for the whole proof) is carried out first. In this setting, we achieve fairness in the following way:

- Applications of WHILE-L are given the cost $c_w = \alpha_w \cdot k_w + o_w$, where k_w is the number of applications of WHILE-L that have already been performed on a proof branch, and $\alpha_w > 0$, o_w are constants. This means that the cost for unwinding a loop a further time grows linearly with the number of earlier loop unwindings.
- Applications of EXPAND-LIST are given the cost $c_e = \alpha_e \cdot k_e + o_e$, where k_e is the sum of the depths to which each of the list metavariables has been expanded on a proof branch. This sum can be computed by considering the constraints C that are attached to formulae $\phi \ll C$ in a sequent that contain list access expressions $L \downarrow i$: one can simply count the occurrences of *cons* in the terms that have to be substituted for the original list metavariables when solving the constraint C.[6]

Good values for the constants α_w, α_e are in principle problem-dependent, but in our experience it is meaningful to choose α_e (a lot) bigger than α_w. When proving the formula (5), yielding the constraint shown in Sect. 4, we had chosen $\alpha_w = 50$, $o_w = 200$, $\alpha_e = 2500$, $o_e = -2000$.

A slightly different approach is to choose a fixed upper bound either for the number of loop unwindings or for the heap size, and to let only the other parameter grow unboundedly within one proof attempt. If the proof attempt fails, the bound can be increased and a new proof is started. In the experiments so far, we have not found any advantages of starting multiple proof attempts over the method described first, however.

5.3 Arithmetic Handling in KeY

The heap representation that is introduced in Sect. 2.3 heavily uses arithmetic (both natural and integer numbers). After the elimination of programs using

[6] The actual computation of c_e is more complicated, because smaller costs are chosen when applying EXPAND-LIST for terms $L \downarrow i$ in which i is a concrete literal, or when the rule has already been applied for the same list L earlier.

symbolic execution, of updates and of list expressions, the construction of solutions or closing constraints essentially boils down to handling arithmetic formulae. Although KeY is in principle able to use the theorem prover Simplify [12] as a back-end for discharging goals that no longer contain modal operators and programs, this does not provide any support when reasoning with metavariables (Simplify does not use metavariables). In this section, we shortly describe the native support for arithmetic that we, thus, have added to KeY.

Linear Arithmetic Equations and inequations over linear polynomials is the most common and most important fragment of integer arithmetic. We use Fourier-Motzkin variable elimination to handle such formulae—inspired by the Omega test [13], which is an extension of Fourier-Motzkin. Although Fourier-Motzkin does not yield a complete procedure over the integers, in contrast to the Omega test, we have so far not encountered the need to create a full implementation of the Omega test.

As a pre-processing step, the equations and inequations of a sequent are always moved to the antecedent and are transformed into inequations $c \cdot x \leq s$ or $c \cdot x \geq s$, where c is a positive number and s is a term. Further, in order to ensure termination, we assume the existence of a well-ordering on the set of variables of a problem and require that x is strictly bigger than all variables in s. Fourier-Motzkin variable elimination can then be realised by the following rule:

$$\frac{\Gamma, c \cdot x \geq s, d \cdot x \leq t, d \cdot s \leq c \cdot t \ \vdash\ \Delta}{\Gamma, c \cdot x \geq s, d \cdot x \leq t \ \vdash\ \Delta}\ \textsc{transitivity} \qquad (c > 0, d > 0)$$

Apart from the rule for eliminating variables from inequations, we also have to provide rules for generating closing constraints (using the constraint language from Sect. 4):

$$\frac{[\, s = t\,]}{\Gamma \ \vdash\ s = t, \Delta} =\mathrm{R} \quad \frac{[\, s \neq t\,]}{\Gamma, s = t \ \vdash\ \Delta} =\mathrm{L} \quad \frac{[\, s > t\,]}{\Gamma, s \leq t \ \vdash\ \Delta} \leq\mathrm{L} \quad \frac{[\, s < t\,]}{\Gamma, s \geq t \ \vdash\ \Delta} \geq\mathrm{L}$$

Non-Linear Arithmetic In order to handle multiplication, division- and modulo-operations that frequently occur in programs, we have also added some support for non-linear integer arithmetic to KeY. Our approach is similar to that of the ACL2 theorem prover [14] and is based on the following rule (together with the rules for handling linear arithmetic):

$$\frac{\Gamma, s \leq s', t \leq t', 0 \leq (s' - s) \cdot (t' - t) \ \vdash\ \Delta}{\Gamma, s \leq s', t \leq t' \ \vdash\ \Delta}\ \textsc{mult-inequations}$$

Often, it is also necessary to perform a systematic case analysis. The rule MULT-INEQUATIONS alone is, for instance, not sufficient to prove simple formulae like $x \cdot x \geq 0$. Case distinctions can be introduced with the following rules:

$$\frac{\begin{array}{l}\Gamma, x < 0 \ \vdash\ \Delta \\ \Gamma, x = 0 \ \vdash\ \Delta \\ \Gamma, x > 0 \ \vdash\ \Delta\end{array}}{\Gamma \ \vdash\ \Delta}\ \textsc{sign-cases} \qquad \frac{\begin{array}{l}\Gamma, s < t \ \vdash\ \Delta \\ \Gamma, s = t \ \vdash\ \Delta\end{array}}{\Gamma, s \leq t \ \vdash\ \Delta}\ \textsc{strengthen}$$

We can now prove $x \cdot x \geq 0$ by first splitting on the sign of x. The rules SIGN-CASES and STRENGTHEN are in principle sufficient to find solutions for arbitrary solvable polynomial equations and inequations. Combined with the rules =R, =L, ≤L, ≥L from above, this guarantees that the calculus can always produce solutions and closing constraints for satisfiable sequents that (only) contain such formulae.

6 Related Work

Proof strategies based on metavariables and backtracking are related to common approaches to test data generation with symbolic execution, see, e.g., [5, 7]. Conceiving the approach as *proving* provides a semantics, but also opens up for new optimisations like backtracking-free proof search. Likewise, linear arithmetic is frequently used to handle branch predicates in symbolic execution, e.g. [15]. This is related to Sect. 4, although constraints are in the present paper not only used for branch predicates, but also for the actual pre- and post-conditions.

As discussed in Sect. 3.1, there is a close relation between ground proof procedures and test data generation using actual program execution. Constructing proofs using metavariables can be seen as exhaustive testing, because the behaviour of a program is examined (simultaneously) for all possible inputs. When using the fairness approach of limiting the size of the initial heap that is described in Sect. 5.2, the method is related to bounded exhaustive testing, because only program inputs up to a certain size are considered.

A technique that can be used both for proving programs correct and incorrect is abstraction-refinement model checking (e.g., [16–18]). Here, the typical setup is to abstract from precise data flow and to prove an abstract version of a program correct. If this attempt fails, usually symbolic execution is used to extract a precise witness for program incorrectness or to increase the precision of the employed abstraction. Apart from abstraction, a difference to the method presented here is the strong correlation between paths in a program (reachability) and counterexamples in model checking. In contrast, our approach can potentially produce classes of pre-states that cover multiple execution paths.

Related to this approach is the general idea of extracting information from failing verification attempts, which can be found in many places. ESC/Java2 [19] and Boogie [20] are verification systems for object-oriented languages that use the prover Simplify [12] as back-end. Simplify is able to derive counterexamples from failed proof attempts, which are subsequently used to create warnings about possible erroneous behaviour of a program for certain concrete situations. Another example is [21], where counterexamples are created from unclosed sequent calculus proofs. Making use of failing proof attempts has the advantage of reusing work towards verification that has already been performed, which makes it particularly attractive for interactive verification systems. At the same time, it is difficult to obtain completeness results and to guarantee that proofs explicitly "fail," or that counterexamples can be extracted. In this sense, our approach is more systematic.

7 Conclusions and Future Work

The development of the proposed method and of its prototypical implementation has been driven by working with (small) examples [22], but we cannot claim to have a sufficient number of benchmarks and comparisons to other approaches yet. It is motivating, however, that our method can handle erroneous programs like in Fig. 1 (and similar programs operating on lists) automatically, which we found to be beyond the capabilities of commercial test data generation tools like JTest [23, 22]. This supports the expectation that the usage of a theorem prover for finding bugs (i) is most reasonable for "hard" bugs that are only revealed when running a program with a non-trivial pre-state, and (ii) has the further main advantage of deriving more general (classes of) counterexamples than testing methods. The method is probably most useful when combined with other techniques, for instance with test generation approaches that can find "obvious" bugs more efficiently.

For the time being, we consider it as most important to better understand the constraint language of Sect. 4 for representing solutions, and, in particular, to investigate the decidability of consistency. Because of the extensive use of lists in Sect. 2.3, it would also be attractive to have constraints that directly support the theory of lists. As explained in Sect. 5.1, such constraints would introduce a notion of *heap isomorphism*, which is a topic that we also plan to address. Further, we want to investigate the combination of backtracking and incremental closure (as sketched in Sect. 3.4). A planned topic that conceptually goes beyond the method of the present paper are proofs about the termination behaviour of programs.

Acknowledgements

We want to thank Wolfgang Ahrendt for many discussions that eventually led to this paper, and Tobias Nipkow for comments on an older version of the paper. Thanks are also due to the anonymous referees for helpful comments and hints.

References

1. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
2. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer-Verlag (2007)
3. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C.: JML Reference Manual. (August 2002)
4. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Logic for Programming, Artificial Intelligence and Reasoning. Volume 4246 of LNCS., Springer (2006) 422–436
5. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7) (1976) 385–394
6. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices **35**(9) (2000) 268–279

7. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings of the Second Conference on Computer Science and Engineering in Linkping, ECSEL (October 1999) 21–28
8. Hähnle, R.: Tableaux and related methods. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume I. Elsevier Science B.V. (2001) 101–178
9. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
10. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proceedings, IJCAR, Siena, Italy. Volume 2083 of LNAI., Springer (2001) 545–560
11. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In Garavel, H., Hatcliff, J., eds.: Proceedings, TACAS. Volume 2619 of LNCS., Springer (2003) 553–568
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3) (2005) 365–473
13. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM Press (1991) 4–13
14. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common lisp. IEEE Trans. Softw. Eng. **23**(4) (1997) 203–213
15. Gupta, N., Mathur, A.P., Soffa, M.L.: Automated test data generation using an iterative relaxation method. In: SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (1998) 231–244
16. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model checker. International Journal on Software Tools for Technology Transfer **2**(4) (2000) 410–425
17. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2) (2003) 203–232
18. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings, PLDI. (2001) 203–213
19. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proceedings, PLDI. (2002) 234–245
20. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T., eds.: Post Conference Proceedings, CASSIS, Marseille. Volume 3362 of LNCS., Springer (2005) 49–69
21. Reif, W., Schellhorn, G., Thums, A.: Flaw detection in formal specifications. In: Proceedings, IJCAR, Siena, Italy. LNAI, Springer (2001) 642–657
22. Shah, M.A.: Generating counterexamples for Java dynamic logic. Master's thesis (November 2005)
23. Parasoft: JTest (2006) `www.parasoft.com/jsp/products/home.jsp?product=Jtest`.