Generating Counterexamples for Java Dynamic Logic

Philipp Rümmer*

June 21, 2005

Abstract

First-Order Dynamic Logic is an extension of First-Order Predicate Logic that enables propositions about programs to be made in a natural way. The adherence of a program to certain properties—like preserving invariants or compliance with pre-/postconditions—can be translated into the statement that a particular formula of Dynamic Logic is valid, which can be proved mechanically using calculi. Accordingly, dealing with programs that violate a formal specification leads to the investigation of invalid formulas. We consider a dynamic logic for Java and describe an approach for proving formulas *invalid* that works by reduction to the validity problem. Furthermore, the method allows us to derive concrete counterexamples for validity, which could be a useful tool for debugging programs or specifications.

1 Introduction

In the area of first-order predicate logic, in most cases deduction is about showing the validity of formulas, and consequently the employed calculi are mostly optimised for the positive case of formulas that *are* in fact valid. When dealing with statements about programs—with formulas that are valid if a program has certain properties—the more common case are erroneous programs and thus invalid formulas, however. Though it is often possible to gain information about the nature of a bug by examining failed proof attempts (of validity) of the invalid formula, the calculi involved are not tailored to this purpose. In general attempts of proving invalid formulas will not even terminate.

This paper is concerned with directly proving that formulas of a dynamic logic for the Java programming language [Bec01] are *invalid*. Program states are in this logic modelled as algebras over first-order vocabularies, which means that proving formulas invalid primarily is a search for algebra operations invalidating the formula. This is in general a higher-order problem and more difficult than proving formulas valid.

For the special case of algebras modelling program states, however, one can make use of the fact that programs can in finitely many execution steps only access finitely many data locations. When only searching for those invalidating states that can really be reached from some well-defined initial state—*finite variants* of the initial state—the problem becomes significantly easier. In this setting algebra operations can be

*Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden, `philipp@cs.chalmers.se`

represented using algebraic datatypes, which in turn can be defined in first-order logic with arithmetic.

Hence, proving a formula invalid can for a certain fragment of the dynamic logic be reduced to proving formulas (of the same logic) valid. From such a proof one can also derive concrete counterexamples for the original formula. When dealing with formulas assuring the correctness of a program, the counterexample shows a state in which the program behaves inconsistently with its specification. Since automation of the method appears to be possible to a large degree, this might result in a practical tool supporting software development. By employing an existing proof system for validity, the KeY system [ABB$^+$05], we can also cover most parts of Java and handle real-world applications.

The following sections make up this paper: Sect. 2 gives a background about validity and satisfiability of formulas. Sect. 3 introduces the considered dynamic logic for Java. Sect. 4 describes the approach for disproving formulas and for searching for counterexamples. The method is illustrated with an example in Sect. 5, and conclusions and related work are given in Sect. 6, 7.

2 Background: Validity and Satisfiability Problems (Classical Logics)

Examining whether a formula is satisfiable (whether its negation is invalid) is usually regarded a more difficult problem than considering its validity. This is supported by the well-known fact that validity in first-order predicate logic (FOL) is semi-decidable and not decidable [Fit96]. In general, the truth of the statement is mostly determined by the underlying notion of semantic *structures* that govern the evaluation of formulas.[1] Given a class $\mathcal{S}$ of such structures, (i) a formula is called *valid* (in $\mathcal{S}$) if and only if it is evaluated to 'true' for all elements $S \in \mathcal{S}$, and (ii) a formula is called *satisfiable* (in $\mathcal{S}$) if there is an $S \in \mathcal{S}$ such that the formula is evaluated to 'true'.

What a structure actually is varies with the kind of the considered logic: for FOL, structures are algebras over first-order vocabularies (i.e. a universe of individuals that is equipped with operations according to the vocabulary), whereas modal logics or dynamic logics are usually given semantics in terms of *Kripke structures* [HKT00].

2.1 Validity and Satisfiability for First-Order Logic

In theorem proving, most commonly the validity of FOL formulas is examined in the context of entirely loose semantics, i.e. $\mathcal{S}$ is a class $\mathcal{S}_{\mathrm{L}}$ of all algebras over a certain vocabulary. The satisfiability of formulas is in this framework a difficult problem (for sufficiently rich vocabularies not semi-decidable) because it involves proving the existence of suitable algebra operations. A restriction to poor vocabularies—that for instance only contain nullary symbols—or to other decidable fragments of FOL in many cases diminishes the expressiveness of the logic unacceptably.

On the contrary, when using FOL as a language for program assertions or specifications, its expressiveness is often augmented by restricting $\mathcal{S}$ to a class $\mathcal{S}_{\mathrm{Pe}}$ of *arith-*

[1] The common distinction between structures and variable assignments is left out in this document. Valuations of variables can always be regarded as parts of structures.

metic structures, which are structures whose universe encloses the natural numbers and that provide the standard operations $0, 1, +, \cdot, \leq, \doteq$ [HKT00]. Particularly interesting for this paper is the special case of formulas that are *exclusively* build from the symbols $0, 1, +, \cdot, \leq, \doteq$, classical propositional junctors and first-order $\forall$, $\exists$-quantification. If closed formulas of this kind are evaluated in the singleton set $\mathcal{S}^0_{\mathrm{Pe}}$ that consists only of the algebra of natural numbers itself, then validity and satisfiability are the same property, i.e. a formula is valid if and only if it is satisfiable. At the same time the logic is theoretically more expressive than FOL with loose semantics. The present document is motivated by this observation and tries to carry over the result to a dynamic logic for the Java programming language [GJSB00]: to prove formulas of the dynamic logic invalid, they are first transformed in a way such that validity and satisfiability become equivalent.

3 Java Dynamic Logic (JavaDL)

First-order dynamic logic (DL) [HKT00] is a multi-modal extension of FOL 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 $\langle p\rangle\varphi$ expresses that φ holds in at least one final state of program p. Box formulas can be regarded as abbreviations $[p]\varphi \equiv \neg\langle p\rangle\neg\varphi$ as usual. The DL formula that is probably shown most often is $\varphi \to \langle p\rangle\psi$ and states, for a deterministic program p, the total correctness of p concerning a precondition φ and a postcondition ψ.

In the sequel, p is a piece of sequential Java code, i.e. a list of Java statements without static directives like class declarations, and hence the logic is called JavaDL [Bec01].

In order to define a formal semantics of the modal operators it is necessary to capture Java memory states as structures (here first-order structures, i.e. algebras are used) over which the formulas φ, ψ can be evaluated. Java programs are then given semantics as partial functions of these structures, i.e. as functions that map pre-state structures to post-state structures.

3.1 Modelling Java States as Algebras: JavaDL Vocabularies

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, i.e. statically correct. In the whole document we assume that such a system is fixed. The context information is employed to derive a multi-sorted first-order vocabulary such that corresponding structures describe Java memory contents. A detailed description of this derivation can be found in [Bec01]; the following two sections summarise the aspects important for this paper.

3.1.1 JavaDL Vocabularies: Sorts

JavaDL is based on a typed first-order logic with subsorts that resemble the type hierarchies of Java. For each declared Java class C the vocabulary contains a sort of the same name. The interpretation of such a sort is later supposed to be a copy of the set $\mathbb{N}$ of natural numbers, i.e. objects of C are identified with numbers. This set represents a reservoir containing both those objects that are already instantiated 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.

Primitive types are covered by distinguished sorts *int*, *boolean*,[2] etc. The sort *int* is used to represent all different integer types of Java with finite range (a full account thereof is [Sch02]), and is semantically considered as the infinite set $\mathbb{Z}$ of mathematical integers. Accordingly, the interpretations of the other sorts are also fixed to appropriate sets. For technical reasons, a sort *nat* that represents natural numbers is declared.

3.1.2 JavaDL Vocabularies: Functions

The sorts *int*, *boolean*, etc. for primitive Java types are equipped with common operations, in particular $0, 1, +, \cdot, \leq, \doteq$ are defined for the integers, and there are constants $\mathtt{TRUE}, \mathtt{FALSE}$ denoting the two boolean values.

In order to enumerate the particular objects of a class C, we introduce function symbols $obj_C : nat \to C$ that are interpreted with bijections. Instance attributes

```
class C { ... T a; ... }
```

are translated to unary function symbols $a : C \to T$ of the same name. Semantically, such a function can be regarded as an infinite array that contains one entry for each object of C. For distinguishing instantiated objects from non-instantiated ones, an implicit attribute `boolean <created>` is added to each class. Altogether, the instance attribute vectors model the heap during program execution.

Local variables (and also class attributes) can simply be modelled by declaring constants (nullary function symbols).

3.2 Semantics of Java Dynamic Logic

Semantics of modal logics is usually based on the notion of *Kripke structures*: structures provide a set of worlds and transition relations between these worlds. While this approach can be used to give meaning to JavaDL formulas, following it would be unnecessarily complicated because the effect of a particular Java program is completely determined by the initial values of variables and attributes. Given a collection of worlds, there would be at most one compatible transition relation, which enables us to use a simpler definition.

The set of all multi-sorted algebras over the vocabulary built in Sect. 3.1 is in the sequel denoted with $\mathcal{S}_{\mathrm{DL}}$. Interpretations of the particular sorts modelling Java are in $\mathcal{S}_{\mathrm{DL}}$ chosen to be constant and as already described, for instance a sort C representing a Java class is always interpreted with the same copy of $\mathbb{N}$. Each element of $\mathcal{S}_{\mathrm{DL}}$ describes a snapshot that might occur during execution of a program (without information about the program itself and the instruction pointer), namely the contents of the heap (values of attributes) and the values of variables.

If $S, S' \in \mathcal{S}_{\mathrm{DL}}$ are two such states, then we write $S \xrightarrow{p} S'$ to denote that the execution of the Java program p—when starting with the memory contents described by S—terminates in state S'. The evaluation of formulas can directly be defined based on this notion of state transition:

[2] JavaDL strictly distinguishes formulas and terms of sort *boolean*; the latter ones are used to gain a consistent mapping of Java expressions to first-order terms.

Definition 1 (JavaDL Semantics): Suppose that $S \in \mathcal{S}_{\mathrm{DL}}$ is a program state. The value $\mathrm{val}_S(\varphi)$ of a JavaDL formula φ as an element of $\{\mathrm{ff}, \mathrm{tt}\}$ is inductively defined by

$$\mathrm{val}_S(\langle p \rangle \varphi) = \mathrm{tt} \quad \text{iff there is } S' \in \mathcal{S}_{\mathrm{DL}}\text{:} \quad S \xrightarrow{p} S',\ \mathrm{val}_{S'}(\varphi) = \mathrm{tt}$$

and as is common for all first-order connectives (see for instance [Fit96]).

A central observation is that adjacent worlds $S, S' \in \mathcal{S}_{\mathrm{DL}}$, i.e. worlds with $S \xrightarrow{p} S'$ for an arbitrary program p, are *finite variants* [HKT00] of each other. This means that S, S' agree on the interpretations of symbols for all but finitely many points, which follows from the fact that a program can in finitely many execution steps only change the values of finitely many data locations.

3.3 Calculi for the Validity Problem in JavaDL

JavaDL contains arithmetic—as a necessary premise for modelling Java—which directly entails that there are no sound and complete (and finitary) calculi for validity. The usual way of coping with this fact is to assume the existence of an oracle handling the arithmetic fragment of a dynamic logic. For JavaDL, this oracle would be required to semi-decide the validity problem for JavaDL formulas that do not contain any modal operators. Adding modal operators with Java programs does not increase the expressiveness of the logic, and it is possible to construct *relatively complete* calculi for JavaDL validity that translate formulas containing modal operators to formulas without. See [HKT00, Pla04] for a treatment of this topic.

4 Counterexamples

According to the definition of validity in the previous sections, a JavaDL formula φ is *invalid* if there is a particular state $S \in \mathcal{S}_{\mathrm{DL}}$ such that $\mathrm{val}_S(\varphi) = \mathrm{ff}$, or equivalently if $\neg\varphi$ is satisfiable. For a formula that is potentially invalid, it is consequently interesting (i) to investigate whether an invalidating structure exists at all, and, more specifically, (ii) to determine such a structure or a set of such structures as counterexample for φ.

Definition 2 (Counterexample): A *counterexample* of a JavaDL formula φ is an algebra $S \in \mathcal{S}_{\mathrm{DL}}$ over the vocabulary built in Sect. 3.1 such that $\mathrm{val}_S(\varphi) = \mathrm{ff}$.

4.1 Derivation of Counterexamples

Because the domains of the algebras that are potential counterexamples for a formula are fixed, an invalidating one is described uniquely by the interpretations of existing function symbols, i.e. by the values of defined attributes and of variables. In the presence of higher-order quantifiers and for a vocabulary containing the functions $f_1, \ldots, f_n$, a formula φ is consequently invalid if and only if the formula

$$\psi \quad := \quad \exists f_1. \ldots \exists f_n. \neg\varphi \tag{1}$$

is valid (or satisfiable, which is equivalent).[3] The values of $f_1, \ldots, f_n$ range over the corresponding possible operations on the algebra domain, and a counterexample for φ is described by matching choices for $f_1, \ldots, f_n$.

[3] The symbols $f_1, \ldots, f_n$ are shamelessly reused and bound instead of introducing higher-order variables for this purpose.

ψ is in general second-order, however, which means that proving its validity can be more difficult than the validity problem for JavaDL: a relatively complete calculus would require a more powerful (or more mysterious) oracle for higher-order formulas. While this could be viable for a large number of practical verification problems, it would not be possible to directly use implementations of JavaDL validity calculi to this end. Thus, the next section proposes a setting in which higher-order quantification can be replaced with first-order quantification over algebraic data types.

4.2 Restriction to a Fragment of JavaDL

In order to reduce the complexity of showing formulas φ invalid, we first alter the concept of a JavaDL structure of Sect. 3.2 slightly. This is possible because the original definition is very general and also admits program states that cannot occur in reality: during the actual execution of a program there will never be infinitely many instantiated objects, and hence only finitely many points of the attribute vectors $a : C \to T$ introduced in Sect. 3.1.2 are really used to store data. A program p that is started in an initial state $S_0 \in \mathcal{S}_{\mathrm{DL}}$ only reaches states $S \in \mathcal{S}_{\mathrm{DL}}$ that are finite variants of S_0.

Instead of taking all (usually uncountably many) algebras $\mathcal{S}_{\mathrm{DL}}$ into account, consequently we only examine the subset $\mathcal{S}^{\mathrm{Fin}}_{\mathrm{DL}} \subseteq \mathcal{S}_{\mathrm{DL}}$ of finite variants of one particular and fixed algebra $S_0 \in \mathcal{S}_{\mathrm{DL}}$. If one is interested in investigating the behaviour of programs in a realistic context, then a suitable choice for S_0 is a structure in which the elements of all attribute vectors are set to some standard value, and in which in particular the value of the implicit attributes `<created>` is always `FALSE` (no objects are instantiated). This is equivalent to requiring that in $\mathcal{S}^{\mathrm{Fin}}_{\mathrm{DL}}$ instance attributes of classes are always assigned some standard value for all but finitely many objects.

The modification of the semantics can be observed by formulas. Based on an initial state S_0 in which no objects are instantiated, and assuming that a Java class C is declared, for instance the formula

$$\neg\, \forall\, o : C.\ \texttt{<created>}(o) \doteq \texttt{TRUE}$$

is valid in $\mathcal{S}^{\mathrm{Fin}}_{\mathrm{DL}}$ but not in $\mathcal{S}_{\mathrm{DL}}$. Formulas like the one shown are not very meaningful when making statements about program states, however, and are consequently regarded as ill-formed and not taken into account: besides other restrictions, proper formulas only talk about existing objects. Sufficient syntactic criteria for well-formedness (for a similar purpose) are for instance discussed in [Pla04].

The finite variant restriction entails that the range of the quantification $\exists f_1, \ldots, \exists f_n.$ in Eq. (1) can be diminished: now it is only necessary to quantify over finitely (but unboundedly) many points of $f_1, \ldots, f_n$, because only those interpretations of the symbols have to be covered that can actually turn up in algebras of $\mathcal{S}^{\mathrm{Fin}}_{\mathrm{DL}}$. In the presence of arithmetic, such a quantification can be performed using first-order quantifiers, for instance by quantification over lists.

The oracles that are employed in Sect. 3.3 for examining the validity of JavaDL formulas are able to handle arithmetic, and can thus be expected also to cover algebraic datatypes (that in theory are not more difficult than arithmetic, as an encoding of datatypes into arithmetic is in the present situation always possible). In order to obtain a relatively complete calculus for the invalidity of JavaDL formulas φ in the described fragment, one can therefore (i) derive the negated and quantified formula of Eq. (1), (ii) replace the higher-order quantification with algebraic datatypes (which is discussed in the next section), and (iii) use a relatively complete calculus to prove the formula valid.

4.3 From Higher-Order Quantifiers to Quantified Updates

In order to assign the functions $f_1, \ldots, f_n$ values that are determined by algebraic data structures, it is necessary to introduce a further operator, the *update operator* [Bec01, Pla04]. Updates are in JavaDL usually employed to memorise the effect of assignments in programs and enable an efficient treatment of the sequentiality inherent in imperative programming languages.

In the simplest case, terms or formulas can be preceded with expressions of the shape $\{f(s_1, \ldots, s_k) := t\}$ that cause one location of a function f to be updated to the value of t. A valid formula is for instance

$$\{f(1) := 1\}\big(1 \doteq f(f(1))\big)$$

When dealing with the quantifiers $\exists f_1. \ldots \exists f_n.$, the number of cells that need to be assigned is unbounded, which necessitates the usage of a more general kind of updates [Rüm05]. *Quantified updates* allow expressions $\{\texttt{for}\ i \bullet f(s_1, \ldots, s_k) := t\}$, where the variable i may occur in the terms $s_1, \ldots, s_k, t$, and can alter the values of unboundedly or infinitely many locations. A valid formula involving a quantified update is

$$\{\texttt{for}\ i : \mathit{int} \bullet f(i) := 2 \cdot i + 1\}\big(f(f(3)) \doteq 15\big)$$

Being equipped with updates, the particular quantifiers $\exists f_k.$ can be replaced with first-order quantification as follows: the only interesting case are symbols $f_k = a$ modelling instance attributes of classes, i.e. symbols $a : C \to T$. As a first solution, we make use of the algebraic datatype of lists, which can be finitely axiomatised by employing arithmetic.[4] We assume that the datatype is provided through a sort *ListOfT* representing lists of T-values, and through an access operator $l \downarrow i$ that returns the ith element of a list l. To circumvent some technicalities, the operator is required to be total and to return the standard value of attribute a (as in Sect. 4.2) when trying to access non-existing components of l (i is too large).

A formula $\exists a.\varphi$ can then be rewritten to

$$\psi \quad := \quad \exists\, l : \mathit{ListOfT}.\, \{\texttt{for}\ i : \mathit{nat} \bullet a(\mathit{obj}_C(i)) := l \downarrow i\}\ \varphi, \tag{2}$$

i.e. the attribute vector a is updated with the values of list l, and with its standard value in all places that are not determined by l. To see that the rewritten quantification has the right effect, one can recall that the evaluation of formulas takes place in algebras $S \in \mathcal{S}_{\mathrm{DL}}^{\mathrm{Fin}}$, i.e. the symbol a is always interpreted with an attribute vector a^S that contains some standard value in almost all places. The quantified update $\{\texttt{for}\ i : \mathit{nat} \bullet a(\mathit{obj}_C(i)) := l \downarrow i\}$ assigns a such a vector,[5] and all possible vectors can on the other hand be reached by choosing an appropriate list l.

---

[4] In theory, it would equivalently be possible to use an encoding of lists into natural numbers.

[5] The update can also be regarded as binding the symbol a within its scope: renaming a to some other symbol (that does not already turn up in φ) does not change the meaning of the formula. This explains why a seems to occur free in ψ but not in $\exists a.\varphi$.

4.4 Searching For Counterexamples Using Free Variables

After having produced a formula like Eq. (2), one is particularly interested in finding satisfying values of the variable l: the lists that l is ranging over represent (parts of) counterexample candidates for the original formula, namely the values of attribute a for finitely many objects. The derivation of the lists can practically be performed by using a tableaux-like calculus with existential free variables [Fit96]. In this setting, l is in the beginning of a proof replaced with a place holder, a free variable L. From a closed proof one can then read off the desired list by looking at the term that was substituted for L. If the substitution that closed the proof is not ground, then it can even be regarded as a description of a class of counterexamples.

4.5 Different Algebraic Datatypes

Lists are the most obvious algebraic datatype for representing the values of instance attributes, but by far not the only possible one. A different choice would be a datatype for partial functions with finite graph, which would suite the task of specifying the values of attributes for finitely many objects more directly. It can be expected that one needs to make a trade-off between (i) types that enable a good representation of natural classes of counterexamples, and (ii) types that make efficient search for counterexamples possible. Lists have some restrictions regarding the first item, because they make a representation of counterexamples modulo permutation of objects difficult.

5 Example

As test-bed for the described approach the KeY prover [ABB+05] was used, which provides an almost complete implementation of a JavaDL calculus (and a complete implementation for the Java variant JavaCard for smart card software). KeY contains a concept of free existential variables (called metavariables) that work well together with the quantification over algebraic datatypes.

The following paragraphs show an example of disproving a JavaDL formula given in the syntax that is used by the KeY system. Context of the example is a class `C`:

```
public class C { public int attr; }
```

and two variables `x`, `y` of type `C`. In this environment we specify that a piece of Java code swaps the values of the integer attribute of two objects by doing a bit of arithmetic:

```
!x=null & !y=null & x.<created>=TRUE & y.<created>=TRUE ->
ex xPre:int. ex yPre:int. ( x.attr = xPre & y.attr = yPre &
  <{ y.attr += x.attr;
     x.attr  = y.attr - x.attr;
     y.attr -= x.attr; }> ( x.attr = yPre & y.attr = xPre ) )
```

The task of finding the bug in the program (or in the specification) is left to the reader for the time being.

In order to show that this formula is invalid, it is negated and quantifiers are added for all relevant symbols, i.e. for the variables `x`, `y` and the attribute `attr`. The two local variables are treated by two quantifiers (resp.) for covering also the possible value `null`.[6] Accesses to list elements are written `al(Cattr,i)`.

```
ex Cattr:ListOfInt.        ex xId:nat.       ex yId:nat.
ex Ccreated:ListOfBoolean. ex xNull:boolean. ex yNull:boolean.
{ for (i:nat) objC(i).attr := al(Cattr, i),
  for (i:nat) objC(i).<created> := al(Ccreated, i),
  x := if (xNull=TRUE) (null) (objC(xId)),
  y := if (yNull=TRUE) (null) (objC(yId)) }
!( !x=null & !y=null & x.<created>=TRUE & y.<created>=TRUE ->
ex xPre:int. ex yPre:int. ( x.attr = xPre & y.attr = yPre &
  <{ y.attr += x.attr;
     x.attr  = y.attr - x.attr;
     y.attr -= x.attr; }> ( x.attr = yPre & y.attr = xPre )))
```

This formula can be proven valid using KeY in a mostly automated fashion (which means that the considered program violates its specification). In order to illustrate what is happening we show the major steps of a proof. The calculus used is a Gentzen-style sequent calculus for JavaDL [Bec01].

First, the quantifiers can be removed by introducing metavariables (place holders `XID`, `YID`, `XNULL`, `YNULL`, `CATTR`, `CCREATED` for the counterexample) and skolem symbols (denoted with `@xPre`, `@yPre`), and after a number of basic simplifications the proof arrives at two goals. The first goal ensures that the counterexample that is searched for is consistent with the precondition of the original formula. The second goal states that the program does not terminate properly, i.e. that it does not terminate or that the postcondition is violated after its termination.

```
==> XNULL = FALSE            & YNULL = FALSE &
    al(CCREATED, XID) = TRUE & al(CCREATED, YID) = TRUE
===============================================================
 al(CATTR, XID) = @xPre,   al(CATTR, YID) = @yPre,
 XNULL = FALSE,            YNULL = FALSE,
 al(CCREATED, XID) = TRUE, al(CCREATED, YID) = TRUE
==> ! { for (i:nat) objC(i).attr := al(CATTR, i),
        for (i:nat) objC(i).<created> := al(CCREATED, i),
        x := if (XNULL=TRUE) (null) (objC(XID)),
        y := if (YNULL=TRUE) (null) (objC(YID)) }
      <{ y.attr += x.attr;
         x.attr  = y.attr - x.attr;
         y.attr -= x.attr; }> (x.attr = @yPre & y.attr = @xPre)
```

In the second sequent it is now possible to work off the Java program by symbolic execution. This leads to two new goals in the end, because a case distinction between `XID=YID` and `!XID=YID` is made: the expressions `x.attr`, `y.attr` in the program can either address the same or different locations.

[6] This could be done in other, perhaps more elegant ways as well, but until now it is not clear which possibility is the most efficient one in practice.

```
==> XNULL = FALSE            & YNULL = FALSE &
    al(CCREATED, XID) = TRUE & al(CCREATED, YID) = TRUE
=============================================================
 @xPre=0, @yPre=0, XID=YID, ...
==> ! al(CATTR, XID) = 0
=============================================================
 @xPre=al(CATTR, XID), @yPre=al(CATTR, YID), !XID=YID, ...
==> ! al(CATTR, YID) + al(CATTR, XID) - al(CATTR, YID)
        = al(CATTR, XID)
```

At this point a counterexample can be constructed by choosing suitable instantiations of the free variables (KeY is able to find these instantiations automatically). The last goal can be closed by the substitution [YID/XID] (XID is substituted for YID) because of the negated equation of the antecedent. To handle the second goal, the equation al(CATTR,XID)=0 can be made wrong by (for instance) choosing CATTR as the list [1] and XID as 0. Finally, the first goal can be handled by substituting FALSE for XNULL, YNULL and [TRUE] for CCREATED. Altogether, the proof is closed by the substitution

[XID/0, YID/0, CATTR/[1], CCREATED/[TRUE], XNULL/FALSE, YNULL/FALSE]

When having a closer look at the program, one can see that the program violates its specification whenever started in a state in which (i) the variables x and y point to the same object, and (ii) the value of x.attr (which is the same as y.attr) is not 0. The counterexample described by the substitution is one particular instance of this situation.

6 Conclusions and Future Work

A method has been presented that uses calculi for the validity problem for Java Dynamic Logic to disprove formulas, and that is also able to construct counterexamples or whole classes of counterexamples. Within a fragment of JavaDL the method provides relatively complete calculi for satisfiability. Only little implementation work has to be invested to make a theorem prover for JavaDL (in our case KeY) search for counterexamples, and the procedure immediately supports full Java if this is the case for the underlying prover.

Our method is particularly suited for showing that programs are partially incorrect, i.e. that there are inputs for which a program terminates but produces wrong results, because in this setting loops can be treated without inductive arguments: for a particular counterexample one can only have a bounded number of loop iterations. Treating total incorrectness is most likely a more difficult problem (at least in practice) which we plan to investigate in the future.

Though the experiments with finding counterexamples by employing the KeY prover and the presented method are promising so far, it remains to be examined how well the method performs for more complex programs. We expect that this will necessitate a number of optimisations of KeY specifically for counterexample search. In the end it would be interesting to compare the performance of our approach with existing techniques for generating test cases. While the approach described here has a very high precision and the ability to construct whole classes of counterexamples, the complexity caused by using a theorem prover and symbolic execution to this end is significant. Techniques like random testing can be expected to find counterexamples with a much lower effort in many cases, but are bound to fail in situations in which counterexamples are only sparsely distributed over the space of all program inputs and hard to find. In practice it should be beneficial to combine these different paradigms for debugging software, which is one further aspect we consider as future work.

7 Related Work

Model finders like Paradox [CS03] are able to derive finite models of conjectures (provided that there are any). Because JavaDL involves arithmetic, model finders are not directly applicable for the derivation of counterexamples as described in this abstract, but could be used in combination with abstraction techniques.

In [Ahr02], models of conjectures are derived by searching for appropriate valuations of loose functions over freely generated algebras. The method is complete but in general not sound, i.e. it is possible that false models are found. Still it might be possible to use the approach in combination with the ideas given in this abstract to extract counterexamples.

Software testing is concerned with finding input data for a given program that reveals erroneous behaviour. The generation of these input records can be guided in different ways, for instance by creating random data [CH00], by exhaustively covering non-isomorphic classes of input data [BKM02, MK01] or by trying to optimise the execution path coverage in some way. For the latter approach, both actual and symbolic execution of programs are employed [GMS98, BCH$^+$04], and also model-checking techniques are used. Related to our approach is [VPK04], which—besides comparing different approaches for using a model-checker (Java PathFinder [VHBP00]) for creating test input sets with maximal code coverage—focuses on a combination of model-checking and symbolic execution to treat branch predicates and preconditions. In contrast to our method, conditions that are symbolically derived are only used to create test input, and are not verified against a specification.

ESC/Java2 [SoSGox] uses a theorem prover to detect violations of (implicit or explicit) assertions that are added to Java code. The prover Simplify is able to derive counterexamples from failed prove attempts, which are subsequently used to create warnings about possible erroneous behaviour of a program for certain concrete situations.

In [RST01], failed proof attempts of the theorem prover KIV [BRS$^+$00] are used to derive counterexamples for theorems concerning algebraic datatypes and dynamic logic. The approach reuses an existing calculus for the validity problem and is not restricted to a fragment of the concerned logic, but apparently there are no completeness results.

Acknowledgements

I would like to thank Wolfgang Ahrendt for many comments and discussions that led to this paper, and Wojciech Mostowski and Tobias Gedell for numerous complains and discussions about unclear paragraphs. Thanks are also due to the anonymous referees for helpful comments.

References

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[Ahr02] Wolfgang Ahrendt. Deductive search for errors in free data type specifications using model generation. In Andrei Voronkov, editor, *Automated Deduction – CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark*, volume 2392 of *LNCS*. Springer-Verlag, 2002.

[BCH+04] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04, Edinburgh)*, pages 326–335. IEEE Computer Society Press, 2004.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer, 2001.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. *SIGSOFT Softw. Eng. Notes*, 27(4):123–133, 2002.

[BRS+00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer, 2000.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[CS03] Koen Claessen and Niklas Sörensson. New techniques that improve macestyle finite model finding. In Peter Baumgartner and Chris Fermueller, editors, *Model Computation - Principles, Algorithms, Applications*, Miami, Florida, July 2003. CADE-19 Workshop.

[Fit96] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, New York, second edition, 1996.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, 2nd edition, 2000.

[GMS98] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. 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*, pages 231–244. ACM Press, 1998.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, 2000.

[MK01] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 22. IEEE Computer Society, 2001.

[Pla04] André Platzer. An object-oriented dynamic logic with updates. Master's thesis, University of Karlsruhe, Department of Computer Science. Institute for Logic, Complexity and Deduction Systems, September 2004. `http://www.functologic.com/logic/Diplomath.pdf`.

[RST01] Wolfgang Reif, Gerhard Schellhorn, and Andreas Thums. Flaw detection in formal specifications. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 642–657. Springer-Verlag, 2001.

[Rüm05] Philipp Rümmer. A language for sequential, parallel and quantified updates of first-order structures. To appear, 2005. To appear.

[Sch02] Steffen Schlager. Behandlung von Integer Arithmetik bei der Verifikation von Java-Programmen. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, May 2002.

[SoSGox] University of Nijmegen Security of Systems Group. ESC/Java2 (Extended Static Checking for Java Version 2) project, . `http://www.sos.cs.ru.nl/research/escjava/index.html`.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE '00: Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 3. IEEE Computer Society, 2000.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.