A Dynamic Logic for Java Card

Bernhard Beckert

University of Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
`i12www.ira.uka.de/~beckert`

Abstract. In this paper, I describe a Dynamic Logic for JAVA CARD and outline a sequent calculus for this logic that axiomatises JAVA CARD. The purpose of the logic is to provide a framework for software verification that can be integrated into real-world software development processes.

1 Introduction

Design principles and goals. The work that is reported in this paper has been carried out as part of the KeY project [1]. The goal of KeY is to enhance a commercial CASE tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. Accordingly, the design principles for the software verification component of the KeY system are:

- The programs that are verified should be written in a "real" object-oriented (OO) programming language.
- The logical formalism should be as easy as possible to use for software developers (that do not have years of training in formal methods).

Java Card. We use JAVA CARD [10, 5] (soon to be replaced by Java 2 Micro Edition, J2ME) as the target programming language. JAVA CARD is a "real" OO language and has, accordingly, features that are difficult to handle such as dynamic data structures, exceptions, and initialisation; but it lacks some crucial complications of the full JAVA language such as threads and dynamic loading of classes. JAVA smart cards are an extremely suitable application for software verification: (a) JAVA CARD applications are small (JAVA smart cards currently offer 32K memory for code); (b) at the same time, JAVA CARD applications are embedded into larger program systems or business processes which should be modeled (though not necessarily formally verified); (c) JAVA CARD applications are often security-critical, giving incentive to apply formal methods; (d) the high number of deployed smart cards constitutes a new motivation for formal verification, as arbitrary updates are not feasible.

Dynamic Logic. We use Dynamic Logic (DL) [6], which is an extension of Hoare logic [3], as the logical basis of the KeY system's software verification component. We believe that this is a good choice because deduction in DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of JAVA CARD.

DL is successfully used in the KIV software verification system [9] for a programming language that is not object-oriented; and Poetzsch-Heffter and Müller's definition of a Hoare logic for a JAVA subset [8] shows that there are no principal obstacles to adapting the DL/Hoare approach to OO languages.

DL can be seen as a modal predicate logic with a modality $\langle p \rangle$ for every program p (we allow p to be any legal JAVA CARD program); $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program p. In classical DL there can be several such states (worlds) because the programs can be non-deterministic; but here, since JAVA CARD programs are deterministic, there is exactly one such world—if p terminates—or there is no such world—if p does not terminate. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds. A formula $\phi \to \langle p \rangle \psi$ is valid if for every state s satisfying precondition ϕ a run of the program p starting in s terminates, and in the terminating state the postcondition ψ holds.

Thus, the formula $\phi \to \langle p \rangle \psi$ is similar to the Hoare triple $\{\phi\} p \{\psi\}$. But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators: In Hoare logic, the formulas ϕ and ψ are pure first-order formulas, whereas in DL they can contain programs. DL allows to involve programs in the descriptions ϕ resp. ψ of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Also, JAVA constructs such as `instanceof` are available in DL for the description of states. It is, therefore, not necessary to define an abstract data type *state* and to represent states as terms of that type (as has, for example, been done in [8]); instead DL formulas can be used to give a (partial) description of states, which is a more flexible technique and allows to concentrate on the relevant properties of a state.

In comparison to classical DL (that uses a simple "artificial" programming language), a DL for a "real" OO programming language like JAVA CARD has to cope with the following complications:

- A program state does not only depend on the value of program variables but also on the values of the attributes of all existing objects.
- The evaluation of a JAVA expression may have side effects; thus, there is a difference between an expression and a logical term.
- Language features such as built-in data types, exception handling, and object initialisation have to be handled.

2 Syntax of Java Card DL

The non-dynamic part of our DL is basically a typed first-order predicate logic. To define its syntax, we have to specify its sets of variables, its types, and its terms (which we often call “logical terms” in the following to emphasise that they are different from Java expressions). Then, we define what the programs of the DL are. In the programs that are part of a DL formula, we allow an extension of Java Card, where logical terms may occur in place of expressions of the same type. Finally, the syntax of DL formulas and sequents is defined.

Context. We do not allow class definitions in the programs that are part of DL formulas, but define syntax and semantics of DL formulas w.r.t. a given Java Card program (the context), i.e., a sequence of class definitions. With the following restrictions any syntactically legal Java Card program may be used: A context must not contain occurrences of *local inner classes*; and `break` and `continue` must be used with (explicit) labels. These restrictions are “harmless” because any Java Card program can easily be transformed accordingly.

We assume that the following methods and fields are implicitly defined for each class *Cls* in the context and can thus be used in DL formulas (but not in the context). They allow to access information about the program state that is otherwise inaccessible in Java: a list of all existing objects of a class and information on whether a class resp. its objects are initialised. The objects of a certain class are considered to be organised into an (infinite) ordered list; this list is used by `new` to “create” objects (intuitively, `new` changes the attributes `lastCreatedObj` of the class and sets the attribute `created` of the new object to `true`, see Section 4).

```
public static Cls firstObj;  // the first object in the list,
                             // whether already created or not
public static Cls lastCreatedObj;  // the last created object,
                                   // null if no object exists
public Cls prevObj;  // the previous object in the list,
                     // null if for the first object
public Cls nextObj;  // the next object in the list
public boolean beforeObj(Cls obj);  // returns true if this
                                    // is before obj in the list
public boolean created;  // true if the object has already been
                         // created with new, and false otherwise
public static boolean classInitialised;  // true if the class resp.
public boolean objInitialised;           // the object is initialised
```

Variables. In classical DL there is only one type of variables. Here however, to avoid confusion, we use two kinds of variables.

Program variables are denoted with `x`, `y`, `z`, … Their value can differ from state to state and can be changed by programs. They occur in programs and can also be used in the non-program parts of formulas (there they behave like modal constants, i.e., constants whose value can differ from state to state). Program variables cannot be quantified and they cannot be instantiated with terms.

Logical variables are denoted with x, y, z, … They are assigned the same values in all states; a statement such as “x `= 1;`”, which tries to change the value of the logical variable x, is illegal. Logical variables must be bound by a quantifier, free occurrences are not allowed; they can be instantiated with terms (preserving syntactical correctness of a formula but not necessarily its satisfiability or validity).

Types. The set of types of our DL contains (a) the primitive types of JAVA CARD (`boolean`, `byte`, `short`), (b) the classes (object types) defined in the context, (c) the built-in classes such as `String`, and (d) an array type for each of the types in (a)–(c). In addition, there are user-defined types; typically these are abstract data types. There is no type hierarchy, i.e., no sub-typing concept.

Terms. Logical terms are constructed as usual from program variables, logical variables, and the constant and function symbols of all types. The set of terms includes in particular all JAVA CARD literals for the primitive types, string literals, and the `null` object reference literal.

In addition, (a) if o is a term of class type C (i.e., denotes an object) and `a` is a field (attribute) of class C, then o`.a` is a term. (b) If *`Class`* is a class name and *`a`* is a static field of *`Class`*, then *`Class.a`* is a term. (c) If a is an array type term and i is a term of type `byte`, then $a[i]$ is a term.

Programs. The programs in DL formulas are executable code; as said above, they are not allowed to contain class declarations. The (basic) programs are the legal JAVA CARD statements, including: (a) expression statements such as "`x = 1;`" (assignments), "`m(1);`" (method calls), "`i++;`", "`new Cls;`", local variable declarations (which restrict the "visibility" of program variables); (b) blocks and compound statements built with `if-else`, `switch`, `for`, `while`, and `do-while`; (c) statements with exception handling using `try-catch-finally`; (d) statements that abruptly redirect the control flow (`throw`, `return`, `break`, `continue`); (e) labelled statements; (f) the empty statement.

The technique for handling method calls in a DL calculus is to syntactically replace the call by the method's implementation. To handle the `return` statement in the right way, it is necessary to record the program variable or attribute that the result is to be bound to and to mark the boundaries of the implementation when it is substituted for the method call. For that purpose, we allow statements of the form `call(`x`=`m`(`arg_1`,...,`arg_n`)){`$prog$`}` to occur in DL programs.

In addition, we allow programs in DL formulas (not in the context) to contain logical terms. Wherever a JAVA CARD expression can be used, a term of the same type as the expression can be used as well. Accordingly, expressions can contain terms (but not vice versa).

Formulas. Formulas are built as usual from the (logical) terms, the predicate symbols of all the types and the equality predicate $\doteq$, the logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, the quantifiers $\forall$ and $\exists$ (that can be applied to logical variables but not to program variables), and the modal operator $\langle p \rangle$, i.e., if p is a program and ϕ is a formula, then $\langle p \rangle \phi$ is a formula as well.

If o is a variable of some class type C, then a quantification such as $(\forall o)\phi(o)$ ranges over the (infinite) set of all objects of type C whether they have been created or not. The fact that all *created* objects of class C have a certain property ϕ can be expressed using the formula $(\forall o)(o.\mathtt{created} \doteq \mathtt{true} \rightarrow \phi(o))$.

To simplify notation, we allow *updates* of the form $\{x \leftarrow t\}$ resp. $\{o.a \leftarrow t\}$ to be attached to terms and formulas, where x is a program variable, o is a term denoting an object with attribute a, and t is a term. The intuitive meaning of an update is that the term or formula that it is attached to is to be evaluated after changing the state accordingly, i.e., $\phi^{\{x \leftarrow t\}}$ has the same semantics as $\langle x = t \rangle \phi$ (but is easier to handle because the evaluation of t is known to have no side effects).

Sequents. A sequent is of the form $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$ $(m, n \geq 0)$, where the ϕ_i and ψ_j are DL formulas. The meaning of a sequent is that the conjunction of the ϕ_i's implies the disjunction of the ψ_j's.

3 Semantics of Java Card DL

To define the semantics of JAVA CARD DL we use the semantics of the JAVA CARD programming language. In case of doubt, we refer to the precise formal semantics of JAVA defined by Börger and Schulte [4] using Abstract State Machines.[1]

The models of DL are Kripke structures consisting of possible worlds that are called states. All states of a model share the same universe containing a sufficient number of elements of each type.

The function and predicate symbols that are not user-defined—such as the equality predicate and the function symbols of the primitive JAVA CARD types—have a fixed interpretation. In all models they are interpreted according to their intended semantics resp. their meaning in the JAVA CARD language.

Logical variables are interpreted using a (global) variable assignment; they have the same value in all states of a model.

States. In each state a (possibly different) value (an element of the universe) of the appropriate type is assigned to: (a) the program variables, (b) the attributes (fields) of all objects, (c) the class attributes (static fields) of all classes in the context, and (d) the special object variable `this`. Variables and attributes of object types can be assigned the special value *null*.

Note, that states do not contain any information on control flow such as a program counter or the fact that an exception has been thrown.

---

[1] Following another approach, Nipkow and von Oheimb have obtained a precise semantics of a JAVA sublanguage by embedding it into Isabelle/HOL; they also use an axiomatic semantics [7].

Programs and Formulas The semantics of a program p is a state transition, i.e., it assigns to each state s the set of all states that can be reached by running p starting in s. Since JAVA CARD is deterministic, that set either contains exactly one state or is empty. The set of states of a model must be closed under the reachability relation for all programs p, i.e., all states that are reachable must exist in a model (other models are not considered).

The semantics of a logical term t occurring in a program is the same as that of an expression whose evaluation is free of side-effects and gives the same value as t.

For formulas ϕ that do not contain programs, the notion of ϕ being satisfied by a state is defined as usual in first-order logic. A formula $\langle p \rangle\phi$ is satisfied by a state s if the program p, when started in s, terminates in a state s' in which ϕ is satisfied. A formula is satisfied by a model M, if it is satisfied by one of the states of M. A formula is valid in a model M if it is satisfied by all states of M; and a formula is valid if it is valid in all models.

We consider programs that terminate abnormally to be non-terminating. Examples are a program that throws an uncaught exception and a `return` statement that is not within the boundaries of a method invocation. Thus, for example, $\langle$`throw x;`$\rangle\phi$ is unsatisfiable for all ϕ. Nevertheless, it is possible to express and (if true) prove the fact that a program p terminates abnormally (and, for example, throws an exception) using a sequence such as

$$\mathtt{e} \doteq \mathtt{null} \vdash \langle \mathtt{try\{}p\mathtt{\}catch\{Exception\ e\}} \rangle (\neg\, \mathtt{e} \doteq \mathtt{null}) \enspace .$$

Sequents. The semantics of a sequent $\phi_1, \ldots, \psi_m \vdash \psi_1, \ldots, \psi_n$ is the same as that of the formula $(\phi_1 \wedge \ldots \wedge \psi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)$.

4 A Sequent Calculus for Java Card DL

In this section we outline the ideas behind the sequent calculus for JAVA CARD DL, and we present some of the basic rules.[2]

The DL rules of our calculus operate on the first *active* command p of a program $\pi p\,\omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try`-`catch` blocks, and beginnings "`call(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the commands `throw`, `return`, `break`, and `continue` that abruptly change the control flow can be handled appropriatly.[3]

---

[2] These are simplified versions of the actual rules. In particular, initialisation of objects and classes is not considered.

[3] In classical DL, where no prefixes are needed, any formula of the form $\langle p\, q \rangle\phi$ can be replaced by $\langle p \rangle\langle q \rangle\phi$. In our calculus, splitting of $\langle \pi p q \omega \rangle\phi$ into $\langle \pi p \rangle\langle q \omega \rangle\phi$ is not possible (unless the prefix π is empty) because πp is not a valid program; and the formula $\langle \pi p\, \omega \rangle\langle \pi\, q\, \omega \rangle\phi$ cannot be used either because its semantics is in general different from that of $\langle \pi p q \omega \rangle\phi$.

Assignment Rule. The assignment rule is the most important rule of the DL calculus:

$$\frac{\Gamma^{\{x \leftarrow c\}},\ x \doteq \mathit{expr}^{\{x \leftarrow c\}} \ \vdash\ \langle \pi\omega \rangle \phi,\ \Delta^{\{x \leftarrow c\}}}{\Gamma\ \vdash\ \langle \pi\ \ x\ \ \mathtt{=}\ \ \mathit{expr}\mathtt{;}\ \ \omega \rangle \phi,\ \Delta} \qquad c \text{ is a new constant} \tag{1}$$

In classical DL, rule (1) is always applicable; here however, we have to impose a restriction: this rule can only be used if the expression expr is a logical term. Otherwise, other rules have to be applied first to evaluate expr (as that evaluation may have side effects). For example, these rules replace the formula $\langle \mathtt{x\ =\ ++i;} \rangle \phi$ by $\langle \mathtt{i\ =\ i+1;\ \ x\ =\ i;} \rangle \phi$.

Moreover, the handling of updates is more difficult in JAVA CARD DL: In classical DL $\phi^{\{x \leftarrow c\}}$ is equivalent to the formula that is constructed from ϕ by syntactically replacing the left side x of the update by the right side c. Now however, because several object variables may refer to the same object, more complex rules have to be used to simplify the result $\phi^{\{o.\mathtt{a} \leftarrow c\}}$ of an update of an object (or class) attribute.

Rule for Creating Objects. The `new` statement is treated by the calculus as if it were implemented as follows (this implementation accesses the fields that are implicitly defined for all classes, see the explanation in Section 2):

```
public static Cls new() {
  if (lastCreatedObj == null)
      lastCreatedObj = firstObj;
  else
      lastCreatedObj = lastCreatedObj.nextObj;
  lastCreatedObj.created = true;
  return lastCreatedObj;
}
```

Rules for Loops. The following rules allow to “unwind” `while` loops. These are simplified versions that only work if (a) cnd is a logical term (and, thus, its evaluation does not have side effects), and (b) prg does not contain a `continue` statement. Similar rules are defined for `do-while` and `for` loops.

$$\frac{\Gamma\ \vdash\ \mathit{cnd} \doteq \mathtt{true},\ \Delta \qquad \Gamma\ \vdash\ \langle \pi\ \ \mathit{prg}\ \ \mathtt{while\,(}\mathit{cnd}\mathtt{)}\,\mathit{prg}\ \ \omega \rangle \phi,\ \Delta}{\Gamma\ \vdash\ \langle \pi\ \ \mathtt{while\,(}\mathit{cnd}\mathtt{)}\,\mathit{prg}\ \ \omega \rangle \phi,\ \Delta} \tag{2}$$

$$\frac{\Gamma\ \vdash\ \mathit{cnd} \doteq \mathtt{false},\ \Delta \qquad \Gamma\ \vdash\ \langle \pi\omega \rangle \phi,\ \Delta}{\Gamma\ \vdash\ \langle \pi\ \ \mathtt{while\,(}\mathit{cnd}\mathtt{)}\,\mathit{prg}\ \ \omega \rangle \phi,\ \Delta} \tag{3}$$

These rules allow to handle loops if used together with induction schemata for the primitive and the user defined types, such as:

$$\frac{\Gamma\ \vdash\ \psi(c),\ \Delta \qquad \Gamma\ \vdash\ (\forall x)(\psi(x) \to \psi(f(x))),\ \Delta}{\Gamma\ \vdash\ (\forall x)\psi(x),\ \Delta} \tag{4}$$

(where the type of x is generated by c and f).

Rules for Handling Exceptions. The following rules allow to handle `try`-`catch`-`finally` blocks and the `throw` statement. Again, these are simplified versions of the actual rules; they are only applicable if (a) *exc* is a logical term (e.g., a program variable), and (b) the statements `break`, `continue`, and `return` do not occur.

$$\frac{\Gamma \vdash \mathit{instanceof}(\mathit{exc}, T) \quad \Gamma \vdash \langle \pi\ \texttt{try\{}e\texttt{=}\mathit{exc}\texttt{;}\ q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle \pi\ \texttt{try\{throw}\ \mathit{exc}\texttt{;}\ p\texttt{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi,\ \Delta} \tag{5}$$

$$\frac{\Gamma \vdash \neg\mathit{instanceof}(\mathit{exc}, T) \quad \Gamma \vdash \langle \pi\ r\texttt{;}\ \texttt{throw}\ \mathit{exc}\texttt{;}\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle \pi\ \texttt{try\{throw}\ \mathit{exc}\texttt{;}\ p\texttt{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi,\ \Delta} \tag{6}$$

$$\frac{\Gamma \vdash \langle \pi\ r\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle \pi\ \texttt{try\{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi,\ \Delta} \tag{7}$$

Rule (5) applies if an exception *exc* is thrown that is an instance of exception class T, i.e., the exception is caught; otherwise, if the exception is not caught, rule (6) applies. Rule (7) applies if the `try` block is empty and, thus, terminates normally.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. Technical Report 2000/4, University of Karlsruhe, Department of Computer Science, Jan. 2000.
2. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS 1523. Springer, 1999.
3. K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 1981.
4. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In Alves-Foss [2], pages 353–404.
5. U. Hansmann, M. S. Nicklous, T. Schäck, and F. Seliger. *Smart Card Application Development Using Java.* Springer, 2000.
6. D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. Elsevier, Amsterdam, 1990.
7. T. Nipkow and D. von Oheimb. Machine-checking the Java specification: Proving type safety. In Alves-Foss [2], pages 119–156.
8. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Proceedings, Programming Languages and Systems (ESOP), Amsterdam, The Netherlands*, LNCS 1576, pages 162–176. Springer, 1999.
9. W. Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, 1995.
10. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Application Programming Interfaces, Draft 2, Release 1.3*, 1998.