A Program Logic for Handling JAVA CARD's Transaction Mechanism

Bernhard Beckert[1] and Wojciech Mostowski[2]

[1] Institute for Logic, Complexity, and Deduction Systems,
University of Karlsruhe, Germany
beckert@ira.uka.de
[2] Chalmers University of Technology, Göteborg, Sweden
Computing Science Department
woj@cs.chalmers.se

Abstract. In this paper we extend a program logic for verifying JAVA CARD applications by introducing a "throughout" operator that allows us to prove "strong" invariants. Strong invariants can be used to ensure "rip out" properties of JAVA CARD programs (properties that are to be maintained in case of unexpected termination of the program). Along with introducing the "throughout" operator, we show how to handle the JAVA CARD transaction mechanism (and, thus, conditional assignments) in our logic. We present sequent calculus rules for the extended logic.

1 Introduction

Overview. The work presented in this paper is part of the KeY project [1, 9]. One of the main goals of KeY is to provide deductive verification for a real world programming language. Our choice is the JAVA CARD language [6] (a subset of JAVA) for programming smart cards. This choice is motivated by the following reasons. First of all JAVA CARD applications are subject to formal verification, because they are usually security critical (e.g., authentication) and difficult to update in case a fault is discovered. At the same time the JAVA CARD language is easier to handle than full JAVA (for example, there is no concurrency and no GUI). Also, JAVA CARD programs are smaller than normal JAVA programs and thus easier to verify. However, there is one particular aspect of JAVA CARD that does not exist in JAVA and which requires the verification mechanism to be extended with additional rules and concepts: the persistency of the objects stored on a smart card in combination with JAVA CARD's transaction mechanism (ensuring atomicity of bigger pieces of a program) and the possibility of a card "rip out" (unexpected termination of a JAVA CARD program by taking the smart card out of the reader/terminal). Since we want to have support for the full JAVA CARD language in the KeY system we have to handle this aspect.

To ensure that a JAVA CARD program is "rip-out safe" we need to be able to specify "strong" invariants—invariants that must hold throughout the whole execution of a JAVA CARD program (except when a transaction is in progress). The KeY system's deduction component uses a program logic, which is a version of

Dynamic Logic modified to handle JAVA CARD programs (JAVA CARD DL) [2,3]. An extension to pure Dynamic Logic to include trace modalities "throughout" and "at least once" is presented in [4]. Here we extend that work and introduce the "throughout" operator to JAVA CARD DL (we do not introduce "at least once" since it is not necessary for handling "rip out" properties). Then we add techniques necessary to deal with the JAVA CARD transaction mechanism (specifically conditional assignments inside the transactions). We present the sequent calculus rules for our extensions. So far we have not implemented the new rules in the KeY system's interactive prover (the implementation for the unextended JAVA CARD DL is fully functional). But considering the extensibility and open architecture of the KeY prover it is not a difficult task.

Related Work. As said above, the work presented here is based on [4], which extends pure Dynamic Logic with trace modalities "throughout" and "at least once". There exist a number of attempts to extend OCL with temporal constructs, see [5] for an overview. In [16] temporal constructs are introduced to the JAVA Modelling Language (JML), but they refer to sequences of method invocations and not to sequences of intermediate program states.

Structure of the Paper. The rest of this paper is organised as follows. Section 2 gives some more details on the background and motivation of our work and some insights into the JAVA CARD transaction mechanism. Section 3 contains a brief introduction to JAVA CARD Dynamic Logic. Section 4 introduces the "throughout" operator in detail and presents sequent calculus rules to handle the new operator and the transaction mechanism. Section 5 shows some of the rules in action by giving simple proof examples and finally Section 6 summarises the paper.

2 Background

The KeY Project. The main goal of the KeY project [1, 9] 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: (1) The specification language should be usable by people who do not have years of training in formal methods. The Object Constraint Language (OCL), which is incorporated into the current version of the Unified Modelling Language (UML), is the specification language of our choice. (2) The programs that are verified should be written in a "real" object-oriented programming language. We decided to use JAVA CARD (we have already stated our reasons for this decision in the introduction).

For verifying JAVA CARD programs, the already mentioned JAVA CARD Dynamic Logic has been developed within the KeY project (Section 3 contains a detailed description of this logic). The KeY system translates OCL specifications into JAVA CARD DL formulas, whose validity can then be proved with the KeY system's deduction component.

Motivation. The main motivation for this work resulted from an analysis of a JAVA CARD case study [11]. In short, the case study involves a JAVA CARD applet that is used for user authentication in a Linux system (instead of a password mechanism). After analysing the application and testing it, the following observation was made: the JAVA CARD applet in question is not "rip-out safe". That is, it is possible to destroy the applet's functionality by removing (ripping out) the JAVA CARD device from the card reader (terminal) during the authentication process. The applet's memory is corrupted and it is left in an undefined state, causing all subsequent authentication attempts to be unsuccessful (fortunately this error causes the applet to become useless but does not allow unauthorised access, which would have been worse).

It became clear that, to avoid such errors, one has to be able to specify (and if possible verify) the property that a certain invariant is maintained at all times during the applet's execution, such that it holds in particular in case of an abrupt termination. Standard UML/OCL invariants do not suffice for this purpose, because their semantics is that if they hold before a method is executed then they hold after the execution of a method. Normally it is not required for an invariant to hold in the intermediate states of a method's execution. To solve this problem, we introduce "strong" invariants, which allow to specify properties about all intermediate states of a program.

For example, the following "strong" invariant (expressed in pseudo OCL) says that we do not allow partially initialised `PersonalData` objects at any point in our program. In case the program is abruptly terminated we should end up with either a fully initialised object or an uninitialised (empty) one:

```
context PersonalData throughout:
  not self.empty implies
    self.firstName <> null and self.lastName <> null and self.age > 0
```

Since the case study was explored in the context of the KeY project, we extended the existing JAVA CARD DL with a new modality to handle strong invariants.

The JAVA CARD Transaction Mechanism. Here we describe the aspects of transaction handling in JAVA CARD relevant to this paper. A full description of the transaction mechanism can be found in [6, 13–15].

The memory model of JAVA CARD differs slightly from JAVA's model. In smart cards there are two kinds of writable memory: persistent memory (EEPROM), which holds its contents between card sessions, and transient memory (RAM), whose contents disappear when power loss occurs, i.e., when the card is removed from the card reader. Thus every memory element in JAVA CARD (variable or object field) is either persistent or transient. The JAVA CARD language specification gives the following rules (this is a slightly simplified view of what is really happening): All objects (including the reference to the currently running applet, `this`, and arrays) are created in persistent memory. Thus, in JAVA CARD all assignments like "`o.attr = 2;`", "`this.a = 3;`", and "`arr[i] = 4;`" have a permanent character; that is, the assigned values will be kept after the card

loses power. A programmer can create an array with transient elements, but currently there is no possibility to make objects (fields) other than array elements transient. All local variables are transient.

The distinction between persistent and transient objects is very important since these two types of objects are treated in a different way by JAVA CARD's transaction mechanism. The following are the JAVA CARD system calls for transactions with their description:

`JCSystem.beginTransaction()` begins an atomic transaction. From this point on, all assignments to fields of objects are executed conditionally, while assignments to transient variables or array elements are executed unconditionally (immediately).

`JCSystem.commitTransaction()` commits the transaction. All conditional assignments are committed (in one atomic step).

`JCSystem.abortTransaction()` aborts the transaction. All the conditional assignments are rolled back to the state in which the transaction started. Assignments to transient variables and array elements remain unchanged (as if there had not been a transaction in progress).

As an example to illustrate how transactions work in practice, consider the fragment of a JAVA CARD program shown on the right. After the execution of this program, the value of `this.a` is still 100 (value before the transaction), while the value of `i` now is 100 (the value it was updated to during the transaction).

```
this.a = 100;
int i = 0;
JCSystem.beginTransaction();
  i = this.a;
  this.a = 200;
JCSystem.abortTransaction();
```

Transactions do not have to be nested properly with other program constructs, e.g., a transaction can be started within one method and committed within another method. However, transactions must be nested properly with each other (which is not relevant for the current version of JAVA CARD, where the nesting depth of transactions is restricted to 1).

The whole program piece inside the transaction is seen by the outside world as if it were executed in one atomic step (considering the persistent objects). By introducing strong invariants we want to ensure the consistency of the persistent memory of a JAVA CARD applet, thus strong invariants will not (and should not) be checked within a transaction—in case our program is terminated abruptly during a transaction, the persistent variables will be rolled back to the state before the transaction was started for which the strong invariant was established.

3 JAVA CARD Dynamic Logic

Dynamic Logic [7, 8, 10, 12] can be seen as an extension of Hoare logic. It is a first-order modal logic with modalities $[p]$ and $\langle p \rangle$ for every program p (we allow p to be any sequence of JAVA CARD statements). In the semantics of these modalities a world w (called state in the DL framework) is accessible from the

current world, if the program p terminates in w when started in the current world. The formula $[p]\phi$ expresses that ϕ holds in *all* final states of p, and $\langle p\rangle\phi$ expresses that ϕ holds in *some* final state of p. In versions of DL with a non-deterministic programming language there can be several such final states (worlds). 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 $\phi \to \langle p\rangle\psi$ 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 post-condition ψ holds. The formula $\phi \to [p]\psi$ expresses the same, except that termination of p is not required, i.e., ψ must only hold *if* p terminates.

3.1 Syntax of JAVA CARD DL

As said above, a dynamic logic is constructed by extending some non-dynamic logic with modal operators of the form $\langle\cdot\rangle$ and $[\cdot]$. The non-dynamic base logic of our DL is a typed first-order predicate logic. We do not describe in detail what the types of our logic are (basically they are identical with the JAVA types) nor how exactly terms and formulas are built. The definitions can be found in [2]. Note that terms (which we often call "logical terms" in the following) are different from JAVA expressions—they never have side effects.

The programs in DL formulas are basically executable JAVA CARD code. However, we introduced an additional construct not available in plain JAVA CARD, whose purpose is the handling of method calls. Methods are invoked by syntactically replacing the call by the method's implementation. To treat the `return` statement in the right way, it is necessary (a) to record the object field or variable x that the result is to be assigned to, and (b) to mark the boundaries of the implementation *prog* when it is substituted for the method call. For that purpose, we allow statements of the form `method_call(`x`){`*prog*`}` to occur. This is a "harmless" extension because the additional construct is only used for proof purposes and never occurs in the verified JAVA CARD programs.

3.2 Semantics of JAVA CARD DL

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 (if p terminates normally) or is empty (if p does not terminate or terminates abruptly).

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 normally 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. Sequents are notated following the scheme $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$ which has the same semantics as the formula $(\forall x_1)\cdots(\forall x_k)((\phi_1 \wedge \ldots \wedge \phi_m) \to (\psi_1 \vee \ldots \vee \psi_n))$, where $x_1, \ldots, x_k$ are the free variables of the sequent.

3.3 State Updates

We allow *updates* of the form $\{x := t\}$ resp. $\{o.a := t\}$ 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., $\{x := t\}\phi$ has the same semantics as $\langle \texttt{x = t;} \rangle \phi$.

3.4 Rules of the Sequent Calculus

Here we only present a small number of rules necessary to get proper intuition of how the JAVA CARD DL sequent calculus works.

Notation. The rules of our calculus operate on the first *active* statement 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-finally` blocks, and beginnings "`method_call(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active statement is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately. The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the JAVA block "`l:{try{ i=0; j=0; }finally{ k=0; }}`", operating on its first active statement "`i=0;`", then the non-active prefix π is "`l:{try{`" and the "rest" ω is "`j=0; }finally{ k=0; }}`".

In the following rule schemata, $\mathcal{U}$ stands for an arbitrary update.

The Rule for `if`. As the first simple example, we present the rule for the `if` statement:

$$\frac{\Gamma,\ \mathcal{U}(b \doteq \mathit{true}) \ \vdash\ \mathcal{U}\langle \pi p \omega \rangle \phi \quad \Gamma,\ \mathcal{U}(b \doteq \mathit{false}) \ \vdash\ \mathcal{U}\langle \pi q \omega \rangle \phi}{\Gamma \ \vdash\ \mathcal{U}\langle \pi\, \texttt{if(}b\texttt{)\{}p\texttt{\}else\{}q\texttt{\}}\, \omega \rangle \phi} \qquad \text{(R1)}$$

The rule has two premisses, which correspond to the two cases of the `if` statement. The semantics of this rule is that, if the two premisses hold in a state, then the conclusion is true in that state. In particular, if the two premisses are valid, then the conclusion is valid. In practice rules are applied from bottom to top: from the old proof obligation new proof obligations are derived. As the `if` rule demonstrates, applying a rule from bottom to top corresponds to a symbolic execution of the program to be verified.

The Assignment Rule and Handling State Updates. The assignment rule

$$\frac{\Gamma \ \vdash\ \mathcal{U}\{\mathit{loc} := \mathit{expr}\}\langle \pi\, \omega \rangle \phi}{\Gamma \ \vdash\ \mathcal{U}\langle \pi\, \texttt{loc = expr;}\, \omega \rangle \phi} \qquad \text{(R2)}$$

adds the assignment to the list of updates $\mathcal{U}$. Of course, this does not solve the problem of computing the effect of an assignment, which is particularly

complicated in JAVA because of aliasing. This problem is postponed and solved by rules for simplifying updates.

The assignment 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 $\langle$`x = ++i;`$\rangle\phi$ with $\langle$`i = i+1; x = i;`$\rangle\phi$.

4 Extension for Handling "Throughout" and Transactions

In some regard JAVA CARD DL (and other versions of DL) lacks expressivity—the semantics of a program is a relation between states; formulas can only describe the input/output behaviour of programs. JAVA CARD DL cannot be used to reason about program behaviour not manifested in the input/output relation. Therefore, it is inadequate for verifying strong invariants that must be valid throughout program execution.

Following [4], we overcome this deficiency and increase the expressivity of JAVA CARD DL by adding a new modality $[\![\cdot]\!]$ ("throughout"). In the extended logic, the semantics of a program is the sequence of all states its execution passes through when started in the current state (its *trace*). Using $[\![\cdot]\!]$, it is possible to specify properties of the intermediate states of terminating and non-terminating programs. And such properties (typically strong invariants and safety constraints) can be verified using the JAVA CARD DL calculus extended with additional sequent rules for $[\![\cdot]\!]$ presented in Section 4.1.

A "throughout" property (formula) has to be checked after every single field or variable assignment, i.e., the sequent rules for the throughout modality will have more premisses and branch more frequently. According to the JAVA CARD runtime environment specification [14], each single field or variable assignment is atomic. This matches exactly JAVA CARD DL's notion of a single update. Thus, a "throughout" property has to hold after every single JAVA CARD DL update. However, additional checks have to be suspended when a transaction is in progress. This will require marking the modality (resp. the program in the modality) with a tag saying that a transaction is in progress, so that different rules apply. Since transactions do not have to be nested properly with other program constructs, enclosing a transaction in a block with a separate set of rules for that kind of block (like the `method_call` blocks) is not possible.

In addition, we have to cover conditional assignments and assignment rollback (after `abortTransaction`) in the calculus. This not only affects the "throughout" modality, but the $\langle\cdot\rangle$ and $[\cdot]$ modalities as well, since rolling back an assignment affects the final program state.

In practice only formulas of the form $\phi \rightarrow [\![p]\!]\phi$ will be considered. If transient arrays are involved in ϕ (explicitly or implicitly), one also has to prove $\phi \rightarrow \langle$`initAllTransientArrays();`$\rangle\phi$, i.e., that after a card rip-out the reinitialisation of transient arrays preserves the invariant.

4.1 Additional Sequent Calculus Rules for the $[\![\cdot]\!]$ Modality

Below, we present the assignment and the `while` rules for the $[\![\cdot]\!]$ modality. Due to space restrictions, we cannot list all additional rules. However, the other loop rules are very similar to the `while` rule, and all other $[\![\cdot]\!]$ rules are essentially the same as for $[\cdot]$—except for the transaction rules which we present in the next subsection.

The Assignment Rule for $[\![\cdot]\!]$. An assignment `loc = expr;` is an atomic program, if *expr* is a logical term (and, in particular, is free of side effects and can be computed in a single step). By definition, its semantics is a trace consisting of the initial state s and the final state $s' = \{loc := val_s(expr)\}s$. Therefore, the meaning of $[\![\texttt{loc = expr;}]\!]\phi$ is that ϕ is true in both s and s', which is what the two premisses of the following assignment rule express:

$$\frac{\Gamma \vdash \mathcal{U}\phi \quad \Gamma \vdash \mathcal{U}\{loc := expr\}[\![\pi\omega]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\pi\ \texttt{loc = expr;}\ \omega]\!]\phi} \tag{R3}$$

The left premiss states that the formula ϕ has to hold in the state s before the assignment takes place. The right premiss says that ϕ has to hold in the state s' after the assignment—and in all states thereafter during the execution of the rest ω of the program.

It is easy to see that using this rule causes some extra branching of the proofs involving the $[\![\cdot]\!]$ modality. This branching is unavoidable due to the fact that the strong invariant has to be checked (evaluated) for each intermediate state of the program execution. However, many of those branches, which do not involve JAVA CARD programs any more, can be closed automatically.

The `while` *Rule for* $[\![\cdot]\!]$. Another essential programming construct, where the rule for the $[\![\cdot]\!]$ modality differs from the corresponding rule for the $[\cdot]$ modality, is the `while` loop. As in the case of the `while` rule for the $[\cdot]$ modality a user has to supply a loop invariant *Inv*. Intuitively, the rule establishes three things: (1) In the state before the loop is executed, some invariant *Inv* holds. (2) If the body of the loop terminates normally (there is no `break` and no exception is thrown but possibly `continue` is used) then at the end of a single execution of the loop body the invariant *Inv* has to hold again. (3) Provided *Inv* holds, the formula ϕ has to hold during and continuously after loop body execution in all of the following cases: (i) when the loop body is executed once and terminates normally, (ii) when the loop body is not executed (the loop condition is not satisfied), and (iii) when the loop body terminates abruptly (by `break`, `continue`, or throwing an exception) resulting in a termination of the whole loop.

Formally, the `while` rule for $[\![\cdot]\!]$ is the following:

$$\frac{\Gamma \vdash \mathcal{U}Inv \quad Inv \vdash \langle\alpha\rangle true,\ [\beta]Inv \quad Inv \vdash [\![\pi\beta\omega]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\pi\ \lambda\texttt{while(}a\texttt{)}\ \texttt{\{}p\texttt{\}}\ \omega]\!]\phi} \tag{R4}$$

where

$\alpha \equiv \texttt{if(}a\texttt{)}\ \texttt{\{}l_{break}\texttt{:}\ \texttt{\{try \{}l_{cont}\texttt{:}\ \texttt{\{}p'\texttt{\}}\ \texttt{abort;\} catch(Exception e)\{\}\}\}}$
$\beta \equiv \texttt{if(}a\texttt{)}\ l_{cont}\texttt{:}\ l_{break}\texttt{:}\ \texttt{\{}p'\texttt{\}}$

In the above rule, λ is a (possibly empty) sequence "$l_1 : \ldots l_n :$ " of labels, and p' is p with (a) every "`continue;`" and every "`continue` l_i`;`" changed to "`break` l_{cont}`;`" and (b) every "`break;`" and every "`break` l_i`;`" changed to "`break` l_{break}`;`". The three premisses establish the three conditions listed above, respectively. When the program p' terminates normally, the `abort` in α is reached and, thus, the formula $\langle\alpha\rangle \mathit{true}$ evaluates to *false* and $[\beta]\mathit{Inv}$ has to be proved. Enclosing program p' in "`if(`a`)` $\ldots$" takes care of both cases, where the loop body is executed (intermediate loop body execution) and where it is not executed (loop exit). They are later in the proof considered separately by applying the rule for `if`.

4.2 Additional Sequent Calculus Rules for Transactions

Additional Syntax. Before presenting the sequent rules for transactions, we first have to introduce some new programming constructs (statements) and transaction markers to JAVA CARD DL.

The three new statements are `bT` (JAVA CARD beginning of a transaction), `cT` (JAVA CARD end of a transaction, i.e., commit), and `aT` (JAVA CARD end of a transaction, i.e., abort). These statements are used in the proof when the transaction is started resp. finished in the JAVA CARD program. The statements are only part of the rules and not the JAVA CARD programming language. Thus for example, when a transaction is started in a JAVA CARD program by a call to `JCSystem.beginTransaction()` the calculus assumes the following implementation of `beginTransaction()`:

```
public class JCSystem {
  private static int _transDepth = 0;
  public static void beginTransaction() throws TransactionException {
    if(_transDepth > 0)
      TransactionException.throwIt(TransactionException.IN_PROGRESS);
    _transDepth++;
    bT;
  }
  ...
```

Thus, when we encounter any of `bT`, `cT` or `aT` in our proof we can assume they are properly used (nested).

The second thing we need is the possibility to mark modalities (resp. the programs they contain) with a tag saying that a transaction is in progress. We will use two kinds of tags and make them part of the inactive program prefix π in the sequent. The two markers are: "TRcommit:"—a transaction is in progress and is expected to be committed (`cT`), and "TRabort:"—a transaction is in progress and is expected to be aborted (`aT`). This distinction is very helpful in taking care of conditional assignments—since we know how the transaction is going to terminate "beforehand" we can treat conditional assignments correspondingly, commit them immediately in the first case or "forget" them in the second case.

Rules for Beginning a Transaction. For each of the three operators ($\langle\cdot\rangle$, $[\cdot]$, $[\![\cdot]\!]$) there is one "begin transaction" rule (the rules for $\langle\cdot\rangle$ and $[\cdot]$ are identical, so we only show one of them):

$$\frac{\Gamma \vdash \mathcal{U}\phi \quad \Gamma \vdash \mathcal{U}[\![\mathsf{TRcommit}\colon \pi\omega]\!]\phi \quad \Gamma \vdash \mathcal{U}[\![\mathsf{TRabort}\colon \pi\omega]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\pi\ \mathtt{bT};\ \omega]\!]\phi} \tag{R5}$$

$$\frac{\Gamma \vdash \mathcal{U}\langle\mathsf{TRabort}\colon \pi\omega\rangle\phi \quad \Gamma \vdash \mathcal{U}\langle\mathsf{TRcommit}\colon \pi\omega\rangle\phi}{\Gamma \vdash \mathcal{U}\langle\pi\ \mathtt{bT};\ \omega\rangle\phi} \tag{R6}$$

In case of the $[\![\cdot]\!]$ operator the following things have to be established. First of all, ϕ has to hold before the transaction is started. Then we split the sequent into two cases: the transaction will be terminated by a commit, or the transaction will be terminated by an abort. In both cases the sequent is marked with the proper tag, so that corresponding rules can be applied later, depending on the case. The $\langle\cdot\rangle$ and $[\cdot]$ rules for "begin transaction" are very similar to $[\![\cdot]\!]$ except that ϕ does not have to hold before the transaction is started.

Rules for Committing and Aborting Transactions. These rules are the same for all three operators, so we only show the $[\![\cdot]\!]$ rules.

The first two rules apply when the expected type of termination is encountered ("TRcommit:" for commit resp. "TRabort:" for abort). In that case, the corresponding transaction marker is simply removed, which means that the transaction is no longer in progress. These are the rules:

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi\omega]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\mathsf{TRcommit}\colon \pi\ \mathtt{cT};\ \omega]\!]\phi} \tag{R7}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi\omega]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\mathsf{TRabort}\colon \pi\ \mathtt{aT};\ \omega]\!]\phi} \tag{R8}$$

We also have to deal with the case where the transaction is terminated in an unexpected way, i.e., a commit is encountered when the transaction was expected to abort and vice versa. In this case we simply use an axiom rule, which immediately closes the proof branch (one of the proof branches produced by the "begin transaction" rule will always become obsolete since each transaction can only terminate by either commit or abort). The rules are the following:

$$\frac{}{\Gamma \vdash \mathcal{U}[\![\mathsf{TRabort}\colon \pi\ \mathtt{cT};\ \omega]\!]\phi} \tag{R9}$$

$$\frac{}{\Gamma \vdash \mathcal{U}[\![\mathsf{TRcommit}\colon \pi\ \mathtt{aT};\ \omega]\!]\phi} \tag{R10}$$

Rules for Conditional Assignment Handling within a Transaction. Finally, we come to the essence of conditional assignment handling in our rules. In case the transaction is expected to commit, no special handling is required—all the assignments are executed immediately. Thus, the rule for an assignment in the scope of $[\![\mathsf{TRcommit}\colon \ldots]\!]$ is the same as the rule for an assignment within $[\cdot]$ (the same holds for all other programming constructs). Note that, even using the $[\![\mathsf{TRcommit}\colon \ldots]\!]$ modality, ϕ only has to hold at the end of the transaction, which is considered to be atomic.

$$\frac{\Gamma \vdash \mathcal{U}\{loc := expr\}[\![\mathsf{TRcommit}\colon \pi\omega]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\mathsf{TRcommit}\colon \pi\ \mathtt{loc = expr};\ \omega]\!]\phi} \tag{R11}$$

In case a transaction is terminated by an abort, all the conditional assignments are rolled back as if they were not performed. If we know that the transaction is going to abort because of a TRabort: marker, we can deliberately choose not to perform the updates to persistent objects as we encounter them. However, we cannot simply skip them since the new values assigned to (fields of) persistent objects during a transaction may be referred to later in the same transaction (before the abort). The idea to handle this, is to assign the new value to a copy of the object field or array element while leaving the original unchanged, and to replace—until the transaction is aborted—references to persistent fields and array elements by references to their copies holding the new value. Note that if an object field to which no new value has been assigned is referenced (and for which therefore no copy has been initialised), the original reference is used.

Making this work in practice requires changing the assignment rule for the cases where a transaction is in progress and is expected to abort (i.e., where the "TRabort:" marker is present). Also the rules for update evaluation change a bit, which changes the semantics of an update as well, see description of the rule below. The following is the assignment rule for the $[\![\cdot]\!]$ modality with the "TRabort:" tag present. The corresponding rules for $\langle\cdot\rangle$ and $[\cdot]$ are the same:

$$\frac{\Gamma \vdash \mathcal{U}\{loc' := expr'\}[\![\mathsf{TRabort:}\ \pi\omega]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\mathsf{TRabort:}\ \pi\ \texttt{loc = expr;}\ \omega]\!]\phi} \tag{R12}$$

As usual *expr* has to be a logical term. To handle objects fields persistent arrays elements, all sub-expressions such as $obj.a_1.arr[e].a_2\ldots$ in *expr* are replaced by $obj.a_1'.arr'[e']'.a_2'\ldots$ in $expr'$ (for object fields the prime denotes a copy of that field and for array access function [] the prime denotes a "shadow" access function that operates on copies of elements of a given array). The first reference *obj* or *arr* (as in $arr[i].a$) in *expr* is not primed, since it is either a local variable, which is not persistent, or the `this` reference, which is not assignable, or a static class reference, like `SomeClass`, which also can be viewed as not assignable. All subexpressions that are local variables are left unchanged in $expr'$. The expression *loc* on the left side of the assignment and the subexpression e are changed into loc' resp. e' in the same way as all the subexpressions in *expr*.

As mentioned, the semantics of an update has to be changed to take care of the cases when a copy of an object's field has not been initialised. In the new semantics, if the value of $obj.a'$ or $arr[i]'$ is referred to in an update but is not known (i.e., there was no such value assigned in the preceding updates) then it is considered to be equal to $obj.a$ or $arr[i]$, respectively.

The assignments to the copies are not visible outside the transaction, where the original values are used again—the effect of a roll-back is accomplished. Each separate transaction has to have its own copies of fields or array elements, so the second encountered transaction can, for example, use $''$, the third one $'''$, etc.

One more thing that we have to handle here is the case when the programmer explicitly defines an array to be transient (the above rule assumes that it was not the case). It is not possible to know beforehand which arrays are transient and which are not, since they are defined to be transient by reference and not

by name. This problem can be treated by adding an extra field to each array (only in the rules) indicating whether the given array is transient or persistent (rules for initialising arrays can set this field). Then for each occurrence of array reference *arr* in *loc* and *expr* in rule (R12) we can split the proof into two cases, following the schema:

$$\frac{\begin{array}{l}\Gamma,\ \mathcal{U}(o.arr'.trans \doteq true) \ \vdash\ \mathcal{U}\{o.arr'[e'] := expr'\}[\![\mathsf{TRabort}\colon \pi\omega]\!]\phi \\ \Gamma,\ \mathcal{U}(o.arr'.trans \doteq false) \ \vdash\ \mathcal{U}\{o.arr'[e']' := expr'\}[\![\mathsf{TRabort}\colon \pi\omega]\!]\phi\end{array}}{\Gamma \ \vdash\ \mathcal{U}[\![\mathsf{TRabort}\colon \pi\ \texttt{o.arr[e] = expr;}\ \omega]\!]\phi} \quad \text{(R13)}$$

The remaining rules for $[\![\mathsf{TRabort}\colon \cdot]\!]$ (i.e., for other programming constructs) are the same as for $[\cdot]$, and the remaining rules for $[\mathsf{TRabort}\colon \cdot]$ and $\langle\mathsf{TRabort}\colon \cdot\rangle$ are the same as if there were no transaction marker.

5 Examples

In the following, we show two examples of proofs using the above rules. The first example shows how the $[\![\cdot]\!]$ assignment and `while` rules are used, the second example shows the transaction rules in action. The formula we are trying to prove in the second example is deliberately not provable and shows the importance of the transaction mechanism when it comes to "throughout" properties.

The proofs presented here may look like tedious work, but most of the steps can be done automatically, in fact the only place where user interaction is required, is providing the loop invariant. The KeY system provides necessary mechanisms to perform proof steps automatically whenever possible.

Example 1. Consider the program p shown on the right. We show that throughout the execution of this program, the strong invariant $\phi \equiv x \geq 2$ holds, i.e., we prove the formula $x \geq 2 \rightarrow [\![p]\!]x \geq 2$. Figure 1 shows the whole proof labelled with applied rules. Here we only point out the most interesting things.

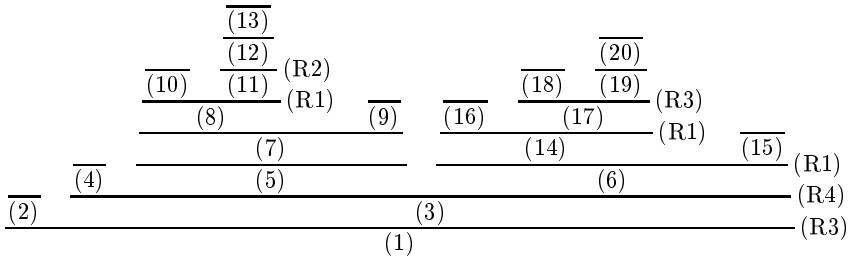
```
x = 3;
while (x < 10) {
  if(x == 2) x = 1;
  else x++;
}
```

When applying the `while` rule (R4) to (3) formula $x \geq 3$ has to be used as the loop invariant *Inv*. Using $Inv' = \phi = x \geq 2$ would not be enough, because the statement `x = 1` inside the `if` statement could not be discarded and x would be assigned 1, which would break the $x \geq 2$ property.

For $x < 10$, the `abort` statement in α is reached after some execution steps (due to space restrictions, we do show the corresponding proof steps). Since `abort` is non-terminating, the formula $\langle\texttt{abort;}\rangle true$ is false and thus (5) can be reduced to (7). All sequents with an empty modality ($[\![\cdot]\!]$ or $[\cdot]$) are reduced by removing the modality; the resulting sequents are then first-order provable. Sequents (9), (10) and (16) are valid by contradiction in the antecedent.

Example 2. Now consider the following program p (fields of o are persistent):

```
bT;
  o.x = 60;
  o.y = 40;
cT;
```



$x \geq 2 \vdash [\![\texttt{x = 3; ...}]\!] x \geq 2$ (1)

$x \geq 2 \vdash x \geq 2$ (2)

$x \geq 2 \vdash \{x := 3\}[\![\texttt{while ...}]\!] x \geq 2$ (3)

$x \geq 2 \vdash \{x := 3\} x \geq 3$ (4)

$x \geq 3 \vdash \langle \alpha \rangle true,\ [\texttt{if(x<10)}\lambda\{\beta\}] x \geq 3$ (5)

$x \geq 3 \vdash [\![\texttt{if(x<10)}\lambda\{\beta\}]\!] x \geq 2$ (6)

$x \geq 3,\ x < 10 \vdash [\texttt{if(x<10)}\lambda\{\beta\}] x \geq 3$ (7)

$x \geq 3,\ x < 10,\ x < 10 \vdash [\lambda\{\beta\}] x \geq 3$ (8)

$x \geq 3,\ x \geq 10,\ x < 10 \vdash [\,] x \geq 3$ (9)

$x \geq 3,\ x < 10,\ x \doteq 2 \vdash [\texttt{x = 1;}] x \geq 3$ (10)

$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash [\texttt{x = x + 1;}] x \geq 3$ (11)

$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash \{x := x + 1\}[\,] x \geq 3$ (12)

$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash x + 1 \geq 3$ (13)

$x \geq 3,\ x < 10 \vdash [\![\lambda\{\beta\}]\!] x \geq 2$ (14)

$x \geq 3,\ x \geq 10 \vdash [\![\,]\!] x \geq 2$ (15)

$x \geq 3,\ x < 10,\ x \doteq 2 \vdash [\![\texttt{x = 1;}]\!] x \geq 2$ (16)

$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash [\![\texttt{x = x + 1;}]\!] x \geq 2$ (17)

$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash x \geq 2$ (18)

$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash \{x := x + 1\}[\![\,]\!] x \geq 2$ (19)

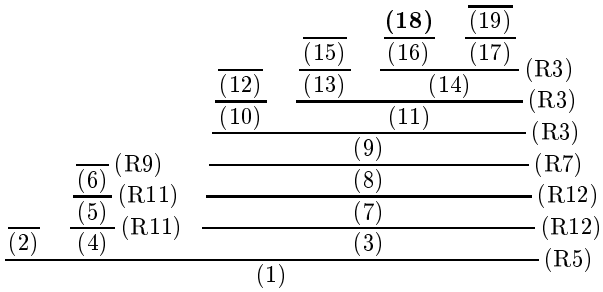
$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash x + 1 \geq 2$ (20)

Abbreviations: $\alpha \equiv$ `if(x < 10) {...`β`; abort; ...}` $\quad \beta \equiv$ `if(x == 2) x = 1; else x++;` $\quad \lambda \equiv l_{cont} : l_{break} :$

Fig. 1. The proof from Example 1

```
t = o.x;
o.x = o.y;
o.y = t;
```

We will try to prove that the strong invariant $o.x + o.y \doteq 100$ holds throughout the execution of this program. Note that this is not provable. The proof attempt is shown in Figure 2. Again, some of the sequents are first-order provable after appropriate reductions. Notice that applying the assignment rule (R11) (resp. (R12)) inside a transaction does not branch. Sequent (6) is proved valid by the axiom rule (R9) (transaction commits unexpectedly). Sequent (18) is not provable. Inspecting our program closely shows that indeed both $o.x$ and $o.y$ are equal to 40 at some point (after `o.x = o.y;` is executed) and their sum is 80, which violates the property we wanted to prove. Thus there is one open proof goal in the proof tree.



$$o.x + o.y \doteq 100 \;\vdash\; [\![\texttt{bT; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{1}$$

$$o.x + o.y \doteq 100 \;\vdash\; o.x + o.y \doteq 100 \tag{2}$$

$$o.x + o.y \doteq 100 \;\vdash\; [\![\mathsf{TRcommit:}\,\texttt{o.x = 60; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{3}$$

$$o.x + o.y \doteq 100 \;\vdash\; [\![\mathsf{TRabort:}\,\texttt{o.x = 60; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{4}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x' := 60\}[\![\mathsf{TRabort:}\,\texttt{o.y = 40; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{5}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x' := 60\}\{o.y' := 40\}[\![\mathsf{TRabort:}\,\texttt{cT; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{6}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}[\![\mathsf{TRcommit:}\,\texttt{o.y = 40; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{7}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}[\![\mathsf{TRcommit:}\,\texttt{cT; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{8}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}[\![\texttt{t = o.x; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{9}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\, o.x + o.y \doteq 100 \tag{10}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\}[\![\texttt{o.x = o.y; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{11}$$

$$o.x + o.y \doteq 100 \;\vdash\; 60 + 40 \doteq 100 \tag{12}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\, o.x + o.y \doteq 100 \tag{13}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\{o.x := o.y\}[\![\texttt{o.y = t; } \ldots]\!]\, o.x + o.y \doteq 100 \tag{14}$$

$$o.x + o.y \doteq 100 \;\vdash\; 60 + 40 \doteq 100 \tag{15}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\{o.x := o.y\}\, o.x + o.y \doteq 100 \tag{16}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\{o.x := o.y\}\{o.y := t\}[\![\,]\!]\, o.x + o.y \doteq 100 \tag{17}$$

$$o.x + o.y \doteq 100 \;\vdash\; 40 + 40 \doteq 100 \tag{18}$$

$$o.x + o.y \doteq 100 \;\vdash\; 40 + 60 \doteq 100 \tag{19}$$

Fig. 2. The proof from Example 2

6 Conclusions and Future Work

We introduced the "throughout" modality (and, thus, strong invariants) to JAVA CARD Dynamic Logic and presented the necessary sequent calculus rules to handle this modality and conditional assignments in JAVA CARD transactions. Intro-

duction of this modality was a manageable task and the set of presented rules is quite easy to use in theorem proving as shown in the examples. Our future plan is to implement our rules in the KeY prover and then try our calculus with "real" examples.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In R.-D. Kutsche and H. Weber, editors, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Grenoble, France*, LNCS 2306, pages 327–330. Springer, 2002.
2. B. Beckert. A dynamic logic for the formal verification of JavaCard programs. In I. Attali and T. Jensen, editors, *Revised Papers, Java on Smart Cards: Programming and Security, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
3. B. Beckert and B. Sasse. Handling Java's abrupt termination in a sequent calculus for Dynamic Logic. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 5–14. TR DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, 2001.
4. B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.
5. J. C. Bradfield, J. K. Filipe, and P. Stevens. Enriching OCL using observational mu-calculus. In R.-D. Kutsche and H. Weber, editors, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Grenoble, France*, LNCS 2306, pages 203–217. Springer, 2002.
6. Z. Chen. *Java Card Technology for Smart Cards.* Addison Wesley, 2000.
7. D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic.* Reidel, 1984.
8. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.
9. KeY project homepage. `http://i12www.ira.uka.de/~projekt/`.
10. D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 14, pages 89–133. Elsevier, 1990.
11. W. Mostowski. Rigorous development of JavaCard applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proc. Fourth Workshop on Rigorous Object-Oriented Methods, London*, 2002. `http://www.cs.chalmers.se/~woj/papers/room2002.ps.gz`.
12. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.
13. Sun Microsystems, Inc. *Java Card 2.2 Application Programming Interface*, 2002.
14. Sun Microsystems, Inc. *Java Card 2.2 Runtime Environment Specification*, 2002.
15. Sun Microsystems, Inc. *Java Card 2.2 Virtual Machine Specification*, 2002.
16. K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In *Algebraic Methodology And Software Technology (AMAST '02)*, LNCS 2422, pages 334–348. Springer-Verlag, 2002.