Formal Reasoning about Non-Atomic Java Card Methods in Dynamic Logic

Wojciech Mostowski

Department of Computing Science, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
W.Mostowski@cs.ru.nl

Abstract. We present an extension to Java Card Dynamic Logic, a program logic for reasoning about Java Card programs, to handle Java Card's so-called non-atomic methods. Although Java Card DL already supports the atomic transaction mechanism of Java Card, non-atomic methods present an additional challenge: state updates triggered by such a non-atomic method are not subjected to any transaction that may possibly be in progress. The semantics of a non-atomic method itself seems to be simple and straightforward to formalise, however experimental studies showed that non-atomic methods affect the whole semantics of the Java Card transaction mechanism in a subtle way, in particular, it affects the notion of a transaction roll-back. In this paper we show how to adapt Java Card DL to accommodate this newly discovered complex transaction behaviour. The extension completes the formalisation of *all* of Java Card in Dynamic Logic.

1 Introduction

Overview. The work we present in this paper can be seen as a final step to complete formalisation of Java Card in Dynamic Logic [2]. Java Card Dynamic Logic (Java Card DL) is a program logic specifically designed to reason about sequential Java programs and, in particular, programs written in Java Card, a Java dialect used to program smart cards. Java Card DL is implemented in the KeY system [1], an integrated design and verification system for object-oriented programs. In an earlier paper we presented an extension to Java Card DL to handle the Java Card's transaction mechanism [3]. The transaction mechanism is a feature specific to Java Card technology. In the context of the persistent data stored in smart card's memory, it allows to ensure that a given program block is executed atomically (to completion or not at all), even when loss of power occurs. The transaction mechanism is deeply embedded in the language specification, i.e. transaction triggering methods in Java Card are native, their implementation cannot be expressed in terms of pure Java code. The support for handling transactions in Java Card DL is important for two reasons: to be able to formally verify atomicity properties in the event of unexpected/premature program termination, and to properly model program state updates caused by transaction roll-back (i.e. *undoing* updates). Indeed, the extended logic allowed

us to prove many interesting properties about real JAVA CARD programs with the KeY system [7,15]. Although we have treated the transaction mechanism thoroughly in our extension to JAVA CARD DL, one particular detail was omitted, namely, two specific *non-atomic* native methods provided by the JAVA CARD API. The intuitive semantics of non-atomic methods seems to be straightforward—a non-atomic method simply excludes the updates it performs from any transaction that might be in progress. However, recent experimental studies [10] show that the intended semantics of JAVA CARD transactions, in particular transaction roll-back, and non-atomic methods is more complex than described in the official JAVA CARD platform documentation [18]. In this paper we present further extensions to JAVA CARD DL to accommodate the extended semantics of the transaction mechanism.

Non-atomic methods are rarely used in JAVA CARD programming, but certain security requirements actually necessitate the use of these methods. An often quoted example concerns PIN try counters. Such a counter is decremented each time a PIN code provided by the user is verified against the code stored on the smart card and the card "shuts down" if too many incorrect guesses are done. By calling the PIN verification routine inside a transaction and deliberately aborting that transaction, the update to the try counter would be rolled back together with all the other updates performed within the transaction. This would be a major security breach, giving a malicious user an infinite number of tries to check PIN validity (the try counter would never be decremented). To avoid such situation a non-atomic method is used to exclude the try counter decrement from the transaction mechanism. Thus, it is really important to be able to reason about non-atomic JAVA CARD methods so that similar security properties can be formally verified. We briefly discuss verification of such a property in Sect. 7.

Related Work. There exist numerous tools and formal systems to reason about JAVA programs on the source code level. Just to name the most important ones: ESC/JAVA2 [6] performs extended static checking of JAVA programs, the LOOP tool [12] employs a Hoare-like logic encoded in higher-order logic (PVS) [11], higher-order logic is also used to formalise a JAVA fragment in Isabelle [19] and in the KRAKATOA tool [13]. The Jive system [14] is based on an extended Hoare style calculus, Jack [4] on weakest precondition calculus, and KIV on yet another version of Dynamic Logic for JAVA CARD [17]. Despite the multiplicity of formal systems designed for JAVA CARD and other "small" JAVA dialects, it seems that (so far) only our framework can truly deal with all of JAVA CARD, including the intricate details of the transaction mechanism. The only other work that investigated JAVA CARD transactions is [9], however the proposed formalism has not been implemented in a tool. The same authors performed experimental studies of the JAVA CARD transaction mechanism [10] that we refer to in this paper.

Structure of the Paper. Sect. 2 and 3 give an overview of the KeY system and JAVA CARD DL, in Sect. 4 we describe the JAVA CARD transaction mechanism and non-atomic JAVA CARD methods. Sect. 5 gives a high-level description of how transactions are treated in JAVA CARD DL, then in Sect. 6 we extend this

description to cover non-atomic methods with a sample of actual calculus rules. Finally, Sect. 7 discusses verification examples and Sect. 8 summarises the paper.

2 The KeY System

Java Card DL has been designed to be the logical infrastructure of the KeY prover. The KeY prover is the core verification component of the KeY system[1] [1]—a tool that enhances a commercial software engineering tool with functionality for formal specification and deductive verification of object oriented programs to be used in real-world software development. Accordingly, the design principles for the software verification component of the KeY system are:

- 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 current version of the Unified Modelling Language (UML), is one specification language that can be used for formal specification. The other language is Java Modelling Language (JML), which has recently become very popular among *formal* Java programmers.
- The programs that are verified should be written in a real object-oriented programming language. The KeY system supports most of sequential Java, and in particular the whole Java Card standard. Since smart card applications are often safety and security critical, Java Card seems to be a perfect target for formal verification.

Our recent research shows that the KeY system performs its job very well—verification of advanced security properties for industrial Java Card applets of non-trivial size is highly feasible and time-wise very efficient [7,15]. In the KeY verification process the OCL or JML specifications are automatically translated into Java Card DL proof obligations, whose validity can then be (in most part automatically) established with the KeY prover. Apart from OCL and JML, Java Card DL can be used explicitly for writing specifications. In the following we briefly describe Java Card DL.

3 Java Card Dynamic Logic

Dynamic Logic [8] 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 (p can be any sequence of Java Card statements). In the semantics of these modalities a state w is accessible from the current state, if the program p terminates in w when started in the current state. 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. Here, since Java Card programs are deterministic, there is exactly one such state (if p terminates) or there is no such state (if p does not terminate).

[1] `http://www.key-project.org`

The formula $\phi \rightarrow \langle p \rangle \psi$ 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 (total correctness). The formula $\phi \rightarrow [p]\psi$ expresses the same, except that termination of p is not required, i.e. ψ must only hold *if* p terminates (partial correctness).

Syntax of JAVA CARD DL As said above, a dynamic logic is constructed by extending some non-dynamic logic with modal operators $\langle \cdot \rangle$ 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 formulae 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.

In order to reduce the complexity of the programs occurring in DL formulae, we introduce the notion of a *program context*. The context consists of API and any additional classes/interfaces used in the program. Syntax and semantics of DL formulae are then defined w.r.t. a given context; the programs in DL formulae are simply blocks of executable JAVA code (method bodies). Programs occurring in DL formulae can also contain special constructs not available in plain JAVA CARD, whose purpose is, among other things, the handling of method calls and the transaction mechanism. For transactions, e.g. JAVA CARD DL recognises special "low-level" transaction statements `bT`, `cT`, and `aT`, which are triggered by the "high-level" API transaction methods `beginTransaction`, etc.

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 formulae 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.

As mentioned above, we consider programs that terminate abruptly to be non-terminating. Thus, e.g. $\langle \texttt{throw x;} \rangle \phi$ is unsatisfiable for all ϕ. Nevertheless, it is possible to express and (if true) prove that a program p terminates abruptly by a simple program transformation. For example, the formula

$$\texttt{exc} = \texttt{null} \rightarrow \langle \texttt{try\{}p\texttt{\}catch(Exception e)\{exc = e;\}} \rangle (\neg(\texttt{exc} = \texttt{null}) \wedge \phi)$$

is true in a state s if and only if the program p, when started in s, terminates abruptly by throwing an exception and condition ϕ is satisfied. The `try-catch` block around program p ensures that the program fragment inside the modality always terminates in a non-abrupt fashion. The postcondition requires p to throw an exception (as otherwise no object is bound to `exc`) and formula ϕ can be established in the abrupt termination state (in fact, this is how JML `signals` clauses are represented in JAVA CARD DL).

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 $(\phi_1 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)$.

Strong Invariants On top of the notion of total (resp. partial) correctness stipulated by the $\langle\cdot\rangle$ (resp. $[\cdot]$) modality, JAVA CARD DL also allows expressing strong invariant properties. A strong invariant specifies that a certain property should be maintained *throughout* the execution of the program (in all the intermediate computation states), which in JAVA CARD DL is expressed with the throughout modality $[\![\cdot]\!]$. This allows one to prove that a given property is preserved even when a premature termination (e.g. card *tear*) of the program occurs. Such properties were the motivation to include support for the transaction mechanism in JAVA CARD DL in the first place [3]. The semantics of the $[\![\cdot]\!]$ modality relies heavily on the notion of atomicity in JAVA CARD, which is affected by non-atomic methods. Because of this, small subtleties are introduced to the formal semantics of $[\![\cdot]\!]$ and a few extra calculus rules are needed. However, in principle, the formalisation for the $[\![\cdot]\!]$ (as well as $[\cdot]$) modality in the context of non-atomic methods does not differ substantially from the formalisation of $\langle\cdot\rangle$, thus, we are not going to discuss the rules for $[\![\cdot]\!]$. We stress though, that we did implement necessary rules for $[\![\cdot]\!]$ and tested them on relevant examples.

State Updates We allow *updates* of the form $\{x := t\}$ resp. $\{o.a := t\}$ to be attached to terms and formulae, 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 $\langle \texttt{x = t;} \rangle \phi$.

Rules of the Sequent Calculus Here we present two sample rules to give the reader 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, e.g. an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch` blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately.[2] 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 following JAVA block

[2] In DL versions for simple artificial programming languages, where no prefixes are needed, any formula of the form $\langle pq \rangle \phi$ can be replaced by $\langle p \rangle \langle q \rangle \phi$ with a sequential composition rule. In our calculus, splitting of $\langle \pi pq\omega \rangle \phi$ 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 $\langle \pi pq\omega \rangle \phi$.

operating on its first active command `i=0;` then the non-active prefix π and the "rest" ω are the marked parts of the block:

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \}catch(Exception e)\{ k=0; \}\}}}_{\omega}$$

In the following rule schemata, $\mathcal{U}$ stands for an arbitrary list of updates, $\mathcal{U}\{u\}$ for an update u appended to $\mathcal{U}$, and $\mathcal{U}\phi$ for ϕ evaluated by applying updates in $\mathcal{U}$.

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

$$\frac{\Gamma,\ \mathcal{U}(b = \mathsf{TRUE})\ \vdash\ \mathcal{U}\langle\pi\, p\,\omega\rangle\phi \qquad \Gamma,\ \mathcal{U}(b = \mathsf{FALSE})\ \vdash\ \mathcal{U}\langle\pi\, q\,\omega\rangle\phi}{\Gamma\ \vdash\ \mathcal{U}\langle\pi\,\texttt{if(b)}\,p\,\texttt{else}\,q\ \omega\rangle\phi} \tag{R1}$$

The rule has two premises, which correspond to the two cases of the `if` statement. The semantics of this rule is that, if the two premises hold in a state, then the conclusion is true in that state. In particular, if the two premises 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. For every Java programming construct there is such a symbolic execution rule, later we explain how transactions are handled rather in terms of symbolic execution, than by discussing all of the relevant calculus rules.

The Assignment Rule and Handling State Updates. The assignment rule:

$$\frac{\Gamma\ \vdash\ \mathcal{U}\{\texttt{loc} := \texttt{expr}\}\langle\pi\,\omega\rangle\phi}{\Gamma\ \vdash\ \mathcal{U}\langle\pi\,\texttt{loc = expr;}\,\omega\rangle\phi} \tag{R2}$$

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 that are attached to formulae whenever possible (without branching the proof). 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 $\langle\texttt{x = i++;}\rangle\phi$ with $\langle\texttt{x = i; i = i + 1;}\rangle\phi$.

4 Java Card Transaction Mechanism

The memory model of Java Card [5,18] 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, in particular, when

the card is removed from the card reader (card *tear*). Thus every memory element in Java Card (variable or object field) is either persistent or transient. Based on the Java Card language specification the following rules can be given:

- 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 permanent character, i.e. the assigned values will be kept after the card loses power.
- A programmer can create an array with transient elements by calling a certain method from the Java Card API (e.g. `JCSystem.makeTransientByteArray`), 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 API calls for transactions:

- `JCSystem.beginTransaction()` begins an atomic transaction. From this point on, all assignments to fields of persistent objects are executed conditionally, while assignments to transient variables or transient array elements are executed unconditionally.[3]
- `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 (at least that is what [18] suggests, we explain what really happens shortly). Assignments to transient data remain unchanged (as if there had not been a transaction in progress).

Considering the persistent objects, the whole program block inside the transaction is seen by the outside world as if it were executed in one atomic step, completely (upon commit), or nor at all (upon abort). A transaction can be aborted explicitly by the programmer, but also implicitly by the Java Card Runtime Environment, when a transaction cannot be completed due to lack of resources or other unexpected program termination (e.g. card *tear*). In the first case, the Java Card program continues its execution with the assignments performed inside the transaction rolled back, in the second case the program is terminated immediately and updates are rolled back during transaction recovery process next time the Java Card applet is initialised. The possibility of an explicit transaction abort has important consequences for the design of the logic to

[3] Terms "conditional and unconditional assignments" that are used in the official Java Card documentation may be a little bit misleading in the context of this work. For our purposes, unconditional assignment should be interpreted as "irreversible" or "immediately permanent", while conditional assignments should be interpreted as "assignments made on copies", so that they can be reverted.

handle transactions; in the logic aborting a transaction can be seen as *undoing assignment* and needs appropriate handling.

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. In the current version of Java Card (2.2.2) the nesting depth of transactions is restricted to 1—only one transaction can be active at a time.

On top of that, Java Card API provides the programmer with two native *non-atomic* methods: `arrayCopyNonAtomic` and `arrayFillNonAtomic` from the `Util` class. In [10], based on extensive experiments performed with Java Card devices, the behaviour of the two methods is thoroughly analysed. Here we only present the highlights that motivated our work.

Methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` copy resp. reset an array, bypassing any transaction that might be in progress, i.e. any changes made to the array will not be rolled-back. We have already motivated the need for exclusion of certain persistent memory locations from the transaction mechanism with the PIN try counter example in the introduction.[4] In the current version, Java Card only allows such non-atomic updates for elements of `byte` arrays, and hence there are only two API methods to take care of non-atomic updates. The consequence for our logic is the following. Apart from committing or aborting, a transaction can also be *suspended* to perform unconditional updates to persistent array elements and later *resumed* to continue updating persistent data conditionally, following the rules of the transaction mechanism again. With the notion of transaction suspension it is also possible to incorporate other non-atomic methods that may appear in future versions of Java Card.

This, however, is not all. The experiments in [10] show that the notion of transaction roll-back is under-specified in the official Java Card documentation [18].[5] Consider two short pieces of Java Card code in Fig. 1. Persistent array `a` stores elements of type `byte` and the `arrayFillNonAtomic` method has the following signature:

```
/** Fill elements off..off+len-1 of bArray with value */
public static void native arrayFillNonAtomic(
  byte[] bArray, short off, short len, byte value);
```

[4] Another reason for introducing native, non-atomic methods for array operations is efficiency, which in the context of this work is not relevant.

[5] Actually, parts of this *under-specification* are deliberate to account for nondeterministic behaviour of some Java Card devices w.r.t. non-atomic methods [10]. Despite this liberal approach there still exist Java Card devices that do not implement non-atomic methods in a correct way—they still go beyond the level of nondeterminism allowed by the official Java Card specification [10]. In our formal model we assume that cards are well behaved (deterministic). Although we present one fixed approach in this paper, the underlying principles of our extension allow us to easily formalise other variants of the transaction model, including a nondeterministic (random) behaviour allowed by the official specification.

```
a[0] = 0;                              a[0] = 0;
beginTransaction();                    beginTransaction();
  a[0] = 1;                              arrayFillNonAtomic(a,0,1,2);
  arrayFillNonAtomic(a,0,1,2);           a[0] = 1;
abortTransaction();                    abortTransaction();
```

Fig. 1. Two transaction roll-back examples

Thus, the call to `arrayFillNonAtomic` in the two examples is equivalent to `a[0] = 2`, with the difference that it bypasses the transaction mechanism. The main difference in the two programs is the value of `a[0]` after the transaction abort. In the program on the left, `a[0]` is rolled-back to 0, the value it was assigned before the transaction was started. In the program on the right, `a[0]` is rolled back to 2, the most recent value it was assigned before the first conditional update happened. To put it in a simpler form, the value to be restored in case of an abort is recorded just before the first conditional update happens, and not when the transaction is started. This is not what the official Java Card documentation would make us believe, it would suggests that the value of `a[0]` should be 0 in both cases. To properly handle the transaction mechanism, our extended logic should distinguish the two situations described above.

5 Symbolic Execution of the Transaction Mechanism

Before we describe how non-atomic methods are handled in Java Card DL, we explain how the basic transaction mechanism is modelled [3]. The main idea is to partly imitate what is actually happening in the Java Card Virtual Machine (or, what we *imagine* is happening); generally, when a transaction is in progress, instead of modifying the original data (unconditional update), the updates are performed on the backup copies of that data (conditional update).

When a call to `beginTransaction` is encountered during the symbolic execution, program analysis (i.e. the proof) is split into two branches. In the first branch the program is analysed with the assumption that the transaction will commit, in the second branch it is assumed that the transaction will be aborted. Later, when an `abortTransaction` statement is encountered on the commit branch, the branch is simply discarded—the symbolic execution is focused on the abort branch. The same exact thing happens in the opposite situation, i.e. when a `commitTransaction` is encountered on the abort branch. On the calculus level, a rule for `beginTransaction` splits the proof into two branches, and each branch (more precisely, the modality containing the program) is marked with an appropriate tag (TRC: or TRA:) saying what kind of transaction finish is expected. Depending on the tag different rules for assignments are applied. Making the distinction between the commit and abort case is very helpful in handling the assignments inside the transaction. On the first branch, since we assume that the transaction is going to commit, we do not have to worry about keeping the backup copies of the modified data, we can commit all the changes as we

encounter them. Conversely, on the abort branch, we know that the assignments eventually (upon encountering `abortTransaction`) will have to be rolled back, so we can choose not to perform them in the first place.[6] Here, however, we encounter a complication: in JAVA CARD only the assignments to the persistent data are rolled back, the assignments to transient data are always performed unconditionally. Moreover, conditionally updated persistent values may be used to update transient variables. Thus, we cannot simply ignore the assignments inside the transactions. Instead, we operate on backup (also called shadow) copies of the persistent data, keeping the original persistent data unmodified, while the updates to transient objects are always performed on the original data.

`this.a = v1;`	`this.a = v1;`
`this.ar[0] = v2;`	`this.ar[0] = v2;`
`int i = 0;`	`i = 0;`
`beginTransaction();`	*The symbolic execution splits into two branches, here we ignore the 'commit' branch, which upon encountering* `abort` *will be discarded. The fact that a transaction was started is recorded.*
`this.a++;`	*First assignment to* `this.a` *inside transaction, create a new backup copy of* `this.a`*:* `this.a' = this.a' + 1;` → `this.a' = this.a + 1;` → `this.a' = v1 + 1;`

`this.ar[0]++;`	`if(this.ar.<transient>)` *// false*
	`this.ar[0] = this.ar[0] + 1; else`
	`this.ar[0]' = this.ar[0]' + 1;` →
	Here, `this.ar[0]'` *also not initialised, use* `this.ar[0]` *on the RHS:*
	`this.ar[0]' = v2 + 1;`

`i = this.a +` `this.ar[0];`	`i` *is local (transient), update unconditionally, on the RHS use already initialised backup copies of* `this.a` *and* `this.ar[0]`*:* `i = this.a' + this.ar[0]';` → `i = v1 + v2 + 2;`
`abortTransaction();`	*Transaction aborted, back to non-transaction mode.*

Fig. 2. Symbolic execution of a JAVA CARD transaction

Let us illustrate this idea with an example in Fig. 2. On the left we give an actual JAVA CARD program, on the right we explain how the symbolic execution (i.e. how the program is interpreted in JAVA CARD DL) of the code on the left proceeds. The prime symbol `'` in combination with the attribute (resp. array element) access operator `.` denotes accessing backup copy of a given attribute (resp. array element) instead of the original value. The arrow → represents subsequent steps in the symbolic execution. If a backup value is required during the evaluation but is not known (has not yet been assigned) the original value is used. The `<transient>` field is assigned to every array object in the JAVA

[6] The two branches correspond to resp. *optimistic* or *pessimistic* approach usually taken in implementing a transaction mechanism.

Card DL model and indicates whether a given array is transient or persistent. Depending on this, the elements of such an array are updated conditionally or unconditionally, following the specification of Java Card transactions. Here we assume that the fact that `this.ar` is persistent (`this.ar.<transient>` is false) is already present in the program analysis. At the end of this symbolic execution it can be established that `i = v1 + v2 + 2`, but the persistent data, `this.a` and `this.ar[0]`, is not affected, the values are equal to `v1` and `v2`, respectively. By performing the assignments on the backup copies, the effect of a transaction roll-back is achieved in the Java Card DL execution model. To sum up, inside a transaction that is assumed to abort, assignments involving persistent data are performed on copies of that data, so that the original values, used again after the abort, remain unchanged. Such specific assignment handling is taken care of specialised Java Card DL calculus rules, which are applied on the (specially tagged for this purpose) abort branch of the proof. The actual rules are described in detail in [3]. In the next section we give a representative sample of these rules updated to accommodate the behaviour of non-atomic methods.

6 Non-Atomic Methods in Java Card DL

Assuming the transaction model just presented we now have to incorporate the semantics of the non-atomic Java Card methods. What `arrayCopyNonAtomic` or `arrayFillNonAtomic` basically do is updating given array elements unconditionally, despite the fact there might be a transaction in progress—the transaction is *suspended* for the time of execution of a non-atomic method. Note, that we cannot assume that the transaction is simply finalised when a non-atomic method is executed, and then a new transaction is started—while the non-atomic method is executed all the conditional assignments executed before the non-atomic method was started are still in effect. When a non-atomic method is finished, the transaction is continued with all the conditional assignments recorded previously. Thus, we introduce the notion of transaction *suspending* and *resuming* to our Java Card DL model.

The symbolic execution model of transactions is affected in the following way. In the commit branch non-atomic methods are not treated in any special way; since we assume that the transaction will commit, it means that all the assignments inside a transaction, including the ones performed by non-atomic methods, will be committed. In the abort branch however, the assignments performed by non-atomic methods should be committed (despite aborted transaction) and all the other assignments should be committed or aborted following the regular Java Card transaction rules. Thus, for the time of execution of a non-atomic method we have to inform the symbolic execution mechanism that the transaction is suspended. The corresponding Java Card DL rule is the following:

$$\frac{\Gamma \vdash \mathcal{U}\langle \mathsf{TRSUSP}{:}\pi\, \omega\rangle\phi}{\Gamma \vdash \mathcal{U}\langle \mathsf{TRA}{:}\pi\, \texttt{suspendTransaction;}\, \omega\rangle\phi} \tag{R3}$$

The meaning of the rule is this: on the abort (TRA: tag) proof branch upon encountering the suspend transaction statement (this statement is only present

in the logic, it is triggered by a call to non-atomic method) mark the branch (modality) with a tag indicating that the transaction is suspended, so that corresponding "non-atomic" assignment rules can be applied. When a non-atomic method is finished, transaction resume statement is triggered and normal transaction processing is again in effect:

$$\frac{\Gamma \vdash \mathcal{U}\langle \mathsf{TRA}{:}\pi\, \omega\rangle\phi}{\Gamma \vdash \mathcal{U}\langle \mathsf{TRSUSP}{:}\pi\, \texttt{resumeTransaction;}\, \omega\rangle\phi} \tag{R4}$$

The idea of symbolic execution of a non-atomic method based on the notion of transaction suspension is illustrated with an example in Fig. 3. As with the previous example, it can be established that after the execution of the program `i = v1 + v2 + 2` and `this.a = v1` (here the update was rolled-back), but the value of `this.ar[0]` is `v2 + 1`—it has been unconditionally updated inside a transaction through transaction suspension that took effect for the time of symbolic execution of `arrayFillNonAtomic`.

`this.a = v1;`	`this.a = v1;`
`this.ar[0] = v2;`	`this.ar[0] = v2;`
`int i = 0;`	`i = 0;`
`beginTransaction();`	*Transaction started, transaction mode for assignments.*
`this.a++;`	`this.a' = this.a' + 1;` → `this.a' = this.a + 1;` → `this.a' = v1 + 1;`
`arrayFillNonAtomic(this.ar, 0, 1, this.ar[0]+1);`	*Transaction is suspended, the code executed by* `arrayFillNonAtomic` *is interpreted as follows. On the LHS update the original (unconditional assignment), on the RHS use backup copies where possible:* `this.ar[0] = this.ar[0]' + 1;` `this.ar[0] = this.ar[0] + 1;` → `this.ar[0] = v2 + 1;`
`i = this.a + this.ar[0];`	*Non-atomic call is finished, resume transaction mode. Here* `this.ar[0]'` *still not initialised, use* `this.ar[0]`*:* `i = this.a' + this.ar[0]';` `i = this.a' + this.ar[0];` → `i = v1 + v2 + 2;`
`abortTransaction();`	*Transaction aborted, back to non-transaction mode.*

Fig. 3. Symbolic execution of a non-atomic method

6.1 Transaction Roll-Back

Finally we have to adapt our model to properly handle transaction roll-back. In Sect. 4 we have already discussed how the values of persistent data are rolled back based on recording values just before the first conditional assignment is executed. Hence, our symbolic execution needs to take care of two more things:

– When an assignment to an array element is done inside a transaction, but not inside a non-atomic method, then we need to record the fact that an array

element has been conditionally assigned. In our model, this information is kept in a boolean array `<trinit>` associated with each array (similarly to the `<transient>` attribute that indicates the persistency type of an array). Unless explicitly initialised, `a.<trinit>[x]` always defaults to false (the update simplification rules of the Java Card DL take care of this).
– When transaction is suspended, before we make an assignment to an array element, we first have to check whether it has been conditionally updated, and depending on the result do a conditional or unconditional assignment.

The corresponding Java Card DL rules are the following. First the rule for the abort branch that records the fact that a given array element has been conditionally assigned and does the actual assignment:

$$\frac{\begin{array}{c}\Gamma,\ \mathcal{U}(\texttt{arr.<transient>} = \mathsf{TRUE})\ \vdash\ \mathcal{U}\{\texttt{arr[e]} := \texttt{expr}'\}\langle \mathsf{TRA}{:}\pi\ \omega\rangle\phi \\ \Gamma,\ \mathcal{U}(\texttt{arr.<transient>} = \mathsf{FALSE})\ \vdash \\ \mathcal{U}\{\texttt{arr.<trinit>[e]} := \mathsf{TRUE}, \texttt{arr[e]}' := \texttt{expr}'\}\langle \mathsf{TRA}{:}\pi\ \omega\rangle\phi\end{array}}{\Gamma\ \vdash\ \mathcal{U}\langle \mathsf{TRA}{:}\pi\ \texttt{arr[e] = expr;}\ \omega\rangle\phi} \tag{R5}$$

For transient arrays, the assignment of the array element is always unconditional (first premise), for persistent arrays (`arr.<transient>` = FALSE), record the information that the given array element has been initialised (`arr.<trinit>[e]` := TRUE) and perform a conditional assignment to that array element (`arr[e]`′ := `expr`′). As with the regular assignment rule (R2), `expr` has to be free of side effects. The prime operator applied to `expr` makes sure that the backup copies are used for all relevant subexpressions occurring in `expr`.

When a transaction is suspended, the rule that takes care of assigning a value to an array element conditionally or unconditionally depending on whether the array element has been already initialised takes the following form:

$$\frac{\begin{array}{c}\Gamma,\ \mathcal{U}(\texttt{arr.<trinit>[e]} = \mathsf{FALSE})\ \vdash\ \mathcal{U}\{\texttt{arr[e]} := \texttt{expr}'\}\langle \mathsf{TRSUSP}{:}\pi\ \omega\rangle\phi \\ \Gamma,\ \mathcal{U}(\texttt{arr.<trinit>[e]} = \mathsf{TRUE})\ \vdash\ \mathcal{U}\{\texttt{arr[e]}' := \texttt{expr}'\}\langle \mathsf{TRSUSP}{:}\pi\ \omega\rangle\phi\end{array}}{\Gamma\ \vdash\ \mathcal{U}\langle \mathsf{TRSUSP}{:}\pi\ \texttt{arr[e] = expr;}\ \omega\rangle\phi} \tag{R6}$$

The interpretation of the rule is this: inside a suspended transaction, if an array element has not yet been conditionally assigned (`arr.<trinit>[e]` = FALSE), update it unconditionally (`arr[e]` := `expr`′), if it has been already conditionally assigned (`arr.<trinit>[e]` = TRUE), keep the assignments conditional (`arr[e]`′ := `expr`′).

These two rules follow the informal description of transactions and non-atomic methods given in Sect. 4. To clarify this, Fig. 4 explains the symbolic execution of the two programs in Fig. 1. Finally, we should note that only the rules for transaction triggering statements and assignments inside a transaction are specific in the context of non-atomic methods, the rules for other programming constructs (e.g. the `if` statement) are the same as in the basic Java Card DL calculus.

`a[0] = 0;`	`a[0] = 0;`
Transaction started, conditional updates.	*Transaction started, conditional updates.*
`a[0]' = 1;` `a.<trinit>[0] = true;`	*Transaction suspended, the execution of* `arrayFillNonAtomic` *unfolds to:* `if(a.<trinit>[0])` *// false* `a[0]'=2; else a[0]=2;` → `a[0]=2;` *Transaction resumed.*
Transaction suspended, the execution of `arrayFillNonAtomic` *unfolds to:* `if(a.<trinit>[0])` *// true* `a[0]'=2; else a[0]=2;` → `a[0]'=2;` *Transaction resumed.*	`a[0]' = 1;` `a.<trinit>[0] = true;`
Transaction aborted, `a[0]` *is 0.*	*Transaction aborted,* `a[0]` *is 2.*

Fig. 4. Symbolic execution for the transaction roll-back

7 Examples

All the rules for the extended Java Card DL to handle non-atomic methods have been implemented in the KeY prover. For the test, we verified the reference implementation of the `check` method from the `OwnerPIN` API class. The property under consideration is the one we mentioned in the introduction: the `check` method should always decrement the try counter (given of course the PIN is not correct and the try counter is not already 0) regardless of any transaction (one about to commit or abort) that might be in progress or any exception that may occur. The Java Card DL formula specifying this is presented in Fig. 5. Since the value of the variable `b` in the program inside the modality is not specified, both possibilities (the transaction will commit or abort) have to be checked, thus we establish the desired property. This formula is proved automatically by the KeY prover in a matter of seconds on a regular Linux desktop computer. Of course, the smaller examples that we have discussed in the paper are also verifiable with the KeY prover. Recall the two programs from Fig. 1. The corresponding Java Card DL formulae (abbreviated) describing their behaviour are presented in Fig. 6—both are quickly discharged by the KeY prover.

8 Summary

We have presented an extension to Java Card Dynamic Logic to handle Java Card non-atomic methods—methods that allow the programmer to exclude updates to persistent data from the transaction mechanism. Although there are only two such methods in the Java Card API, they are of critical importance when certain security issues for smart card applications are considered, as we argued based on the PIN try counter example. Although many people have focused on program verification for Java Card as interesting, small but real, language [12,9,13,14,4,17], Java Card DL with the extension we have presented

```
JCSystem.transactionDepth = 0 ∧ ¬(pin = null) ∧ ¬(pin._triesLeft = null) ∧
... rest of the OwnerPIN basic class specification (class invariant)
_triesLeft@pre = pin._triesLeft[0] ∧ result = −1 →
  ⟨try {
     JCSystem.beginTransaction();
       if(pin.check(pin,offset,length)) result = 1; else result = 0;
     if(b) JCSystem.abortTransaction();
     else  JCSystem.commitTransaction();
   }catch(Exception ex) {}⟩(
      (_triesLeft@pre = 0 → result = 0 ∧ pin._triesLeft[0] = 0) ∧
      (_triesLeft@pre > 0 → (result = 0 →
          pin._triesLeft[0] = _triesLeft@pre − 1)))
```

Fig. 5. Java Card DL specification of the `check` method

```
¬(a = null) ∧ a.<transient> = FALSE →
  ⟨a[0] = 0;
   beginTransaction();
     a[0] = 1;
     arrayFillNonAtomic(a,0,1,2);
   abortTransaction();⟩(a[0] = 0)
```

```
¬(a = null) ∧ a.<transient> = FALSE →
  ⟨a[0] = 0;
   beginTransaction();
     arrayFillNonAtomic(a,0,1,2);
     a[0] = 1;
   abortTransaction();⟩(a[0] = 2)
```

Fig. 6. Java Card DL specifications for the two transaction roll-back examples

here is the first complete program logic for all of Java Card. Due to space restrictions we only discussed a small, but representative sample of the actual Java Card DL calculus rules, however the whole set of rules to deal with non-atomic methods will soon be available in [16]. All the rules have been implemented in the KeY prover and we showed examples of programs that can be verified (automatically) using the extended logic.

Acknowledgements This research is supported by the research program Sentinels (`http://www.sentinels.nl`). Sentinels is financed by the Technology Foundation STW, the Netherlands, Organisation for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs. We would also like to thank anonymous reviewers and Erik Poll for their helpful comments.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, February 2005.
2. B. 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.

3. B. Beckert and W. Mostowski. A program logic for handling JAVA CARD's transaction mechanism. In M. Pezzè, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2003, Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.
4. L. Burdy, A. Requet, and J.-L. Lanet. JAVA applet correctness: A developer-oriented approach. In *Proceedings, Formal Methods Europe 2003*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.
5. Z. Chen. *JAVA CARD Technology for Smart Cards: Architecture and Programmer's Guide*. JAVA Series. Addison-Wesley, June 2000.
6. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for JAVA. In *Proceedings, ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
7. R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In G. Barthe and M. Huisman, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.
8. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
9. E. Hubbers and E. Poll. Reasoning about card tears and transactions in JAVA CARD. In *Fundamental Approaches to Software Engineering (FASE'2004), Barcelona, Spain*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004.
10. E. Hubbers and E. Poll. Transactions and non-atomic API calls in JAVA CARD: Specification ambiguity and strange implementation behaviours. Deptartment of Computer Science NIII-R0438, Radboud University Nijmegen, 2004.
11. M. Huisman and B. Jacobs. JAVA program verification via a Hoare logic with abrupt termination. In *Proceedings, Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 284–303. Springer, 2000.
12. B. Jacobs and E. Poll. JAVA program verification at Nijmegen: Developments and perspective. In *Software Security – Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4–6, 2003. Revised Papers*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.
13. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVA CARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.
14. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000, Berlin, Germany*, volume 1785 of *LNCS*, pages 63–77. Springer, April 2000.
15. W. Mostowski. Formalisation and verification of JAVA CARD security properties in Dynamic Logic. In M. Cerioli, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2005, Edinburgh, Scotland*, volume 3442 of *LNCS*, pages 357–371. Springer, April 2005.
16. W. Mostowski. *The KeY Book*, chapter 9. From Sequential JAVA to JAVA CARD. Springer, 2006. To appear.
17. K. Stenzel. A formally verified calculus for full JAVA CARD. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proceedings, Algebraic Methodology and Software Technology 2004, Stirling, Scotland*, volume 3116 of *LNCS*. Springer, July 2004.
18. Sun Microsystems, Inc., Santa Clara, California, USA. *JAVA CARD 2.2.1 Runtime Environment Specification*, Oct. 2003.
19. D. von Oheimb. *Analyzing JAVA in Isabelle/HOL*. PhD thesis, Institut für Informatik, Technische Universität München, January 2001.