A Case Study in Formal Verification using Multiple Explicit Heaps

Wojciech Mostowski

Formal Methods and Tools, University of Twente, The Netherlands
w.mostowski@utwente.nl

Abstract. In the context of the KeY program verifier and the associated Dynamic Logic for Java we discuss the first instance of applying a generalised approach to the treatment of memory heaps in verification. Namely, we allow verified programs to simultaneously modify several different, but possibly location sharing, heaps. In this paper we detail this approach using the Java Card atomic transactions mechanism, the modelling of which requires two heaps to be considered simultaneously – the basic and the transaction backup heap. Other scenarios where multiple heaps emerge are verification of real-time Java programs, verification of distributed systems, modelling of multi-core systems, or modelling of permissions in concurrent reasoning that we currently investigate for KeY. On the implementation side, we modified the KeY verifier to provide a general framework for dealing with multiple heaps, and we used that framework to implement the formalisation of Java Card atomic transactions. Commonly, a formal specification language, such as JML, hides the notion of the heap from the user. In our approach the heap becomes a first class parameter (yet transparent in the default verification scenarios) also on the level of specifications.

1 Introduction

In the formal verification of object-oriented programs the verification tools and associated logics are constantly improved and developed to handle new verification challenges and to deal with larger and more complex programs. Some of these challenges are efficient reasoning about linked data structures [7, 14] or concurrent programs [11, 15, 1]. Central in all these efforts is the notion of the object heap that is used in respective logics to represent the memory that programs operate on and to handle possible object aliasing. In particular, Separation Logic [22], that some of the verification systems utilise, is strictly built around the notion of the heap, rather than the program that operates on it.

In a similar spirit, the KeY system[1] [2], an interactive verifier for Java programs, was recently redesigned and reimplemented to introduce explicit heap representation to the Java Dynamic Logic [23]. Previously, the heap was represented in the KeY logic implicitly through special semantics of object field

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

updates with complex built-in rewrite rules [2, Chap. 3]. In the new version the heap is explicit, directly accessible through a dedicated program variable `heap`. In particular, this change in heap treatment enables reasoning with dynamic frames [12] in KeY. To accommodate dynamic frames style of specifications KeY uses an extended version of the Java Modelling Language (JML) [4], named JML*. In the proof obligation generation process the framing conditions expressed in JML* are translated to Java Dynamic Logic by constructing appropriate formulae over the heap program variable.

In the default scenario a Java program operates on just one main heap and so does the reasoning system when such programs are verified. Any specification elements, like the JML assignable clauses that express framing conditions, implicitly refer to this one heap. For example, "`assignable o.f, o.g;`" states that fields `f` and `g` of object `o` might be modified on the heap. There are, however, scenarios with computation models that refer to more than one heap. The first and the simplest example is in distributed programs, where some method may modify a set of locations locally as well as remotely through a remote call. For example, such a method could have this framing specification:

```
assignable_local o.f, o.g;
assignable_remote o.g, o.h;
```

stating that an object `o` that is stored both locally and remotely (by holding a copy) is modified partly here and partly there. Note that this notion of multiple heaps is different from Separation Logic, which also talks about several heaps. In Separation Logic a heap can be split into two or more separate heaps with *disjoint* locations. Here we consider heaps that may (but do not have to) share common locations, i.e., one heap may be a (partial) copy of another. In particular, one location can be changed simultaneously on two or more heaps. Generally, different heaps represent different memory sites, either real physical ones or ones introduced to the reasoning system for modelling certain properties.

Our particular use case for considering more than one heap emerged during the rework of the support for Java Card atomic transactions [25, Chap. 7] in KeY to follow the new explicit heap model. The Java Card technology [6] provides a platform to program smart cards with a considerably stripped off version of Java with no concurrency, floating point numbers, or dynamic class loading. The lack of these features along with the security sensitive application areas of smart cards in the financial sector (bank cards), telecommunications (SIM cards), or identity (e.g., electronic passports), used to make Java Card an ideal verification target for many verification tools [24, 16, 5]. However, one complicating factor in Java Card that was initially *overseen* by researchers is the said atomic transaction mechanism. The KeY system was the first verification tool to fully formalise the details of Java Card transactions and provide a working implementation [3, 18].

In short, the transaction mechanism provides a way to group assignments into atomic blocks to preserve the consistency of heap data, which by default in Java Card physically resides in the permanent EEPROM memory. Furthermore, it provides mechanisms to make exceptions to the transaction data roll-back rules as well as to change the default target memory for heap data to reside in

the volatile RAM instead. The core of our formalisation of this in Java Dynamic Logic is the simultaneous use of two heap variables in the computation model. The first represents the regular heap. The second one is used to store the backup copy of the heap for the case when the transaction needs to be aborted and the contents of the heap restored. The assignment rules in the logic operate on both heaps at the same time, raising the need to specify framing conditions for these two heaps in JML*. In effect, the heap becomes a specification parameter, a simple example for heap-parametric frame specification would be:

```
assignable[heap] o.f, o.g;
assignable[backupHeap] o.f;
```

In the remainder of the paper we explain these ideas using the formalisation of the Java Card transaction mechanism in KeY as a case study. This consists of the core formalisation discussed in Sect. 2 and the extensions necessary for modular reasoning discussed in Sect. 3. The use of two heaps is not the only possible way to formalise Java Card transactions. However, the solution with two heaps provides a very clean formalisation with little implementation overhead (discussed in Sect. 4), especially compared to our previous work [3], and gives a uniform framework to apply our ideas in other verification domains. We discuss this in Sect. 5. Finally, we conclude the paper in Sect. 6. The rest of this section is the relevant background information about Java Card [6, 25] and the Java Dynamic Logic [2, 23].

Java Card. The Java Card technology provides means to program smart cards with Java. The technology consists of a language specification, which defines the subset of permissible Java in the context of smart cards, a Virtual Machine specification, which defines the semantics of the Java byte-code when run on a smart card, and finally the API, which provides access to the specific routines usually found on smart cards. The complicating feature of Java Card is that programs directly operate on two memories built into the card chip. Any data allocated in the EEPROM memory is *persistent* and kept between card sessions, the data that resides in RAM is *transient* and always lost on card power-down. The memory allocation rules are: (i) all local variables are transient, (ii) all newly created objects and arrays are by default persistent, and (iii) when allocated with a dedicated API call any array (but not an object) can be made transient. Note the important difference between a reference to an object and the actual object contents. While the object fields are stored in the persistent memory, the reference to that object can be kept in a local variable and be transient itself.[2] Any Java variable, once allocated in its target memory, is transparent to the programmer from the syntax point of view, and it is only the underlying Java Card VM that takes appropriate actions according to the memory type associated with the object.

Objects allocated in EEPROM provide the only permanent storage to an application. To maintain consistency of data in EEPROM, Java Card offers the

[2] A garbage collector is not obligatory in Java Card either. Careless handling of references actually leads to memory leaks, something that is often addressed in Java Card programming guidelines [21].

atomic transaction mechanism accessed through the API. The following is a brief, but complete summary of the transaction rules. Updating a single object field or array element is always atomic. Updates can be grouped into transaction blocks, a static API call to `beginTransaction` opens such a block, which is ended by a `commitTransaction` call, an explicit `abortTransaction` call, or an implicit abort caused by an unexpected program termination (e.g., card power loss). A commit guarantees that all the updates in the block are executed in one atomic step. An abort reverts the contents of the *persistent memory* to the state before the transaction was entered. Note that an explicit abort does not terminate the whole application, only cancels out persistent updates from within the transaction and the program continues its execution. Finally, the API provides so-called *non-atomic* methods to bypass the transaction mechanism. A non-atomic update of a *persistent* array element is never cancelled out by an abort, provided the same array was not manipulated with regular assignments earlier in the same transaction. We provide illustrative examples for these rules later in Sect. 2.

Java Dynamic Logic with Explicit Heap. The Java Dynamic Logic (JDL) [2, 23] of the KeY system is an instance of Dynamic Logic [8] tailored to Java. Modalities $\langle p\rangle\phi$ and $[p]\phi$ represent the notion of total and partial correctness, respectively, of program p w.r.t. property ϕ. Java programs are deterministic, hence total correctness requires p's termination, including the absence of top-level exceptions, for partial correctness termination is not required. Formula ϕ is built from logic terms using the usual connectives. Terms contain references to *logic* variables – rigid symbols whose valuation is independent of the program state, and *program* variables – non-rigid symbols that are program state dependent.

The verification of Java programs in the KeY system is based on symbolic execution realised through a sequent calculus. Program p in the modality is transformed by dedicated calculus rules to progressively reduce the program p into a description of the resulting program state. This description, denoted by $\mathcal{U}$, is called an update, and is essentially a set of canonical assignments of terms to program variables. The following two rules are characteristic for JDL:

$$\frac{\Gamma, \mathcal{U}\mathtt{b} \models \mathcal{U}\langle\pi\, p\, \omega\rangle\phi \quad \Gamma, \mathcal{U}!\mathtt{b} \models \mathcal{U}\langle\pi\, q\, \omega\rangle\phi}{\Gamma \models \mathcal{U}\langle\pi\, \mathtt{if(b)\{}p\mathtt{\}else\{}q\mathtt{\}}\, \omega\rangle\phi}\ \mathsf{if} \qquad \frac{\Gamma \models \mathcal{U}\{\mathtt{v} := \mathtt{se}\}\langle\pi\, \omega\rangle\phi}{\Gamma \models \mathcal{U}\langle\pi\, \mathtt{v=se;}\, \omega\rangle\phi}\ \mathsf{assign}$$

In these rules π denotes an inactive prefix of the program, e.g., a `try{` block opening, a label, or a logic-only description of the current method call stack. The remaining statements of the verified program that the current rule does not operate on are denoted with ω. The if rule unfolds the `if`-statement, which is removed from the modality, and two proof branches are created, where the execution of the two `if` branches, resp. p and q, can continue. The branch condition `b` is evaluated in the current state by applying the state update $\mathcal{U}$ to it. The assign rule transforms an assignment of a simple expression `se` to a local variable `v` into the update $\mathcal{U}$.

A complete symbolic execution of a program results in an empty modality and a set of updates that can be applied to the formula ϕ to check the validity

of the initial claim $\langle p \rangle \phi$. If we consider $\mathcal{U}$ to be an operator on ϕ then it actually is another modality, one that only accepts sequences of canonical assignments as valid programs with the important property that the valuation of the formula ϕ can be quickly performed with a sequence of one-way update simplification and application rules, i.e., an equivalent of the weakest precondition calculus.

So far this covers only local Java variables, the need to reason about objects and arrays introduces the notion of a heap into the logic. The heap is represented as a dedicated program variable of a logic sort *Heap* and treated on equal grounds with other program variables. In particular, references and updates to the heap variable can directly appear in the set of updates $\mathcal{U}$. The *immutable* terms of the sort *Heap* are built using rigid function symbols *select* and *store*, that allow, respectively, querying the heap for a value of a given location, and constructing a new heap with some location updated to a new value w.r.t. some old heap. In particular, the assignment rule for updating an object field `f` is the following (fields are also first class citizens in JDL):

$$\frac{\Gamma \models \mathcal{U}\{\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, \mathtt{se})\}\langle \pi\, \omega \rangle \phi}{\Gamma \models \mathcal{U}\langle \pi\, \mathtt{o.f=se};\, \omega \rangle \phi}\ \mathsf{assignField}$$

For a specification language the KeY system employs JML*, an extension of JML [4] to accommodate dynamic frames [12]. This extension introduces the primitive type of location sets into JML and allows the assignable clauses to refer to variables of such a type instead of static locations. Since dynamic frames are an orthogonal issue to our formalisation of transactions, JML* is synonymous with JML for the work we present here. Moreover, only very basic JML constructs, that we assume the reader is familiar with, are discussed in the paper. In the verification process the KeY system translates a single Java method to be verified and the associated JML* specification into a JDL formula. In this process the heap variable is treated in a special way – it is the properties over this variable that need to be expressed to reflect any framing conditions specified in JML*. A very similar process is applied with similar implications on the heap variable when JML* specifications are used as axioms to replace method calls following the modular verification principles.

The JDL offers other strong facilities for reasoning about Java programs, e.g., the modelling of static initialisation, or comprehensive treatment of Java arithmetic including overflow. However, the work we present in this paper neither affects nor is affected by these other features of the logic. The KeY system itself is a GUI based user-friendly interactive verifier for JDL with a high degree of automation to minimise unnecessary interaction, often leading to fully automatic proofs even for considerably complex programs and properties.

2 Java Card Transactions on Explicit Heaps

In the following, driven by examples, we gradually present the complete formalisation of the Java Card transaction semantics in the KeY JDL and show how multiple heap variables are used. To start with, we introduce native transaction

statements to the Java syntax handled by the logic. That is, the logic should allow for the symbolic execution of `#beginTr`, `#commitTr`, and `#abortTr` that define the transaction boundaries in the verified program. Bridging the actual transaction calls from the API to these statements is a straightforward extension of the verification system. Then, consider the snapshot (slightly artificial on purpose) of a Java Card program on the right, where the fields `balance` and `opCount` of object **this** are persistent, permanently storing the current balance and operation count of some payment application. The local variables `change` and `newBalance` are transient. Ignoring the transaction statements for the moment, the symbolic execution of this program results in the following state updates:

```
int newBalance = 0;
#beginTr;
this.opCount++;
newBalance =
  this.balance + change;
if(newBalance < 0) {
  #abortTr;
}else{
  this.balance = newBalance;
  #commitTr;
}
```

$$\begin{aligned}
&\texttt{newBalance} := 0,\\
&\texttt{heap} := \mathit{store}(\texttt{heap}, \textbf{this}, \texttt{opCount}, \mathit{select}(\texttt{heap}, \textbf{this}, \texttt{opCount}) + 1),\\
&\texttt{newBalance} := \mathit{select}(\texttt{heap}, \textbf{this}, \texttt{balance}) + \texttt{change},\\
&\texttt{heap} := \mathit{store}(\texttt{heap}, \textbf{this}, \texttt{balance}, \texttt{newBalance}) \quad (\text{when } \texttt{newBalance} \geq 0)
\end{aligned}$$

The symbolic execution of the **if** statement splits the proof, so the last update only appears on the **else** proof branch where $\texttt{newBalance} \geq 0$ is assumed.

After further simplification, this set of state updates can be applied to evaluate a property querying e.g., the value of operation count, which in the logic would be the term $\mathit{select}(\texttt{heap}, \textbf{this}, \texttt{opCount})$. The result would indicate a one unit increase w.r.t. the value stored on the heap before this code is executed.

Basic Transaction Roll-back Assuming a simplified Java Card definition, updates to local variables should be kept, while the updates to persistent locations should be rolled back to the state before the transaction was started. The persistent locations in the actual program are synonymous with the data stored on the heap in the logic. Hence, in the first attempt it should be sufficient to roll back the value of the whole heap. This can be done by introducing two simple rules for transaction statements `#beginTr` and `#abortTr` that, respectively, store and restore the value of the heap to and from a backup heap variable `bHeap`:

$$\frac{\Gamma \models \mathcal{U}\{\texttt{bHeap} := \texttt{heap}\}\langle \pi\, \omega\rangle \phi}{\Gamma \models \mathcal{U}\langle \pi\, \texttt{\#beginTr}; \omega\rangle \phi}\ \textsf{begin} \qquad \frac{\Gamma \models \mathcal{U}\{\texttt{heap} := \texttt{bHeap}\}\langle \pi\, \omega\rangle \phi}{\Gamma \models \mathcal{U}\langle \pi\, \texttt{\#abortTr}; \omega\rangle \phi}\ \textsf{abort}$$

This can be done and works as expected because the `heap` variable as modelled in KeY JDL has *call by value* characteristics. Now the set of state updates (on the negative `newBalance` branch) of our example program is the following:

$$\begin{aligned}
&\texttt{newBalance} := 0,\\
&\texttt{bHeap} := \texttt{heap},\\
&\texttt{heap} := \mathit{store}(\texttt{heap}, \textbf{this}, \texttt{opCount}, \mathit{select}(\texttt{heap}, \textbf{this}, \texttt{opCount}) + 1),\\
&\texttt{newBalance} := \mathit{select}(\texttt{heap}, \textbf{this}, \texttt{balance}) + \texttt{change},\\
&\texttt{heap} := \texttt{bHeap} \quad (\text{when } \texttt{newBalance} < 0)
\end{aligned}$$

Whatever terms referring to heap contents should be evaluated with this set of updates, the result would be the values on the heap at the point where it was saved in the `bHeap` variable. The commit statement needs no special handling apart from silent stepping over this statement. In this case the saved value of the `heap` in the `bHeap` variable is simply forgotten until a possible subsequent new transaction where `bHeap` is freshly overwritten with a more recent `heap`.

For the very superficial treatment of transaction semantics this is enough to model transactions in JDL. Note that, so far, no new or assignment rules of any kind were introduced and the new heap variable `bHeap` is not modified in any way apart from being initialised to hold a complete copy of the regular heap.

Transaction Marking and Balancing The two rules we just introduced do not enforce any order on the transaction statements, they allow to successfully verify malformed programs like "`#abortTr; #beginTr;`" or "`#commitTr; #commitTr;`". Furthermore, by Java Card specification, transactions cannot be nested, i.e., the maximum allowed transaction depth is 1, attempts to exceed this limit cause a run-time exception. On the other hand, the scope of a single transaction is very liberal according to the specification – a transaction can be in progress for as long as the card session is active, regardless of the stack of method calls. To simplify our formalisation, we opt for enforcing a stronger requirement – a transaction should be contained in one single Java method. That is, any method that opens a transaction has to close it before the method terminates. This does not exclude complete methods to be called during a transaction, but it does exclude a transaction opening in one method, and closing in another one that is eventually called later on. Our requirement is justified by Java Card security guidelines [21] that ban programs with transaction blocks spanning over several methods (to prevent transaction buffer overruns).[3] In practice, our formalisation not only relies on this requirement, but also enforces it, i.e., programs not adhering to this requirement do not verify.

Consequently, our formalisation restricts the transaction scope in the following way. A transaction marker *TR* attached to a modality indicates that the current execution context of the verified program is an open transaction. Rules for handling transaction opening and closing statements are now sensitive to this marker and automatically enforce correct transaction balancing. Similarly, rules for discharging empty modalities prevent closing proofs with a remaining transaction marker. In turn, any transaction block has to appear in a single verification context (modality), i.e., one method. Furthermore, the dedicated rule for array assignments can be singled out for transaction contexts only. This keeps verification of regular Java programs clear of any unnecessary transaction

[3] Following a similar security rationale we disallow object allocation inside transactions. Real Java Card programs cause serious security risks when objects are allocated in transactions [20], while the formalisation to deal with the "shady" semantics of object deallocation mandated by the Java Card specification [25] would require modelling of explicit garbage collection, something that Java verification systems in principle are not designed for.

artefacts in the proofs. Finally, knowing that the current point in the symbolic execution is a transaction context is important in modular verification for the local interpretation of heap parametric specifications as explained later in Sect. 3.

The rules for transaction statements are the following. An explicit rule for the commit statement is added, in which nothing happens to the `heap` variable, but the transaction context is cancelled out by removing the TR marker:

$$\frac{\Gamma \models \mathcal{U}\{\texttt{bHeap} := \texttt{heap}\}\langle {}_{TR}\pi\, \omega\rangle\phi}{\Gamma \models \mathcal{U}\langle \pi\, \texttt{\#beginTr}; \omega\rangle\phi}\ \textsf{begin}$$

$$\frac{\Gamma \models \mathcal{U}\{\texttt{heap} := \texttt{bHeap}\}\langle \pi\, \omega\rangle\phi}{\Gamma \models \mathcal{U}\langle {}_{TR}\pi\, \texttt{\#abortTr}; \omega\rangle\phi}\ \textsf{abort} \qquad \frac{\Gamma \models \mathcal{U}\langle \pi\, \omega\rangle\phi}{\Gamma \models \mathcal{U}\langle {}_{TR}\pi\, \texttt{\#commitTr}; \omega\rangle\phi}\ \textsf{commit}$$

Persistent and Transient Arrays So far in our formalisation we roll back the whole contents of the backup heap, i.e., we operate the `bHeap` variable as a whole without changing single object locations on it. The separate transaction treatment for the persistent and transient arrays in Java Card now requires also selectively modifying the backup heap, as we describe in the following.[4]

The Java Card transaction rules require that the contents of transient arrays, allocated by dedicated API methods, are never rolled back. Since in JDL all arrays are stored on the heap, we somehow need to introduce a *selective* roll-back mechanism. We achieve this with the following. Whenever an array element is updated in a transaction we check for the persistency type of the array. The check itself is done by introducing an additional implicit boolean field to all objects, called `<transient>`, that maintains the information about the object's persistency type. Standard allocation rules set this field to false, while the dedicated API methods for creating transient arrays specify this field to be true.

Then, when handling assignments, for persistent arrays we take no additional action, for transient arrays we update the value on the heap and *simultaneously* update the value on the backup heap `bHeap`. During an abort, the regular heap is restored to the contents of the backup heap that now also includes updates to transient arrays that were not supposed to be rolled back. The core of the resulting assignment rule for arrays is the following:

$$\frac{\begin{array}{l}\Gamma, \mathcal{U}\texttt{!a.<transient>} \models \mathcal{U}\{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{a}, \texttt{i}, \texttt{se})\}\langle {}_{TR}\pi\, \omega\rangle\phi \\ \Gamma, \mathcal{U}\texttt{a.<transient>} \models \mathcal{U}\{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{a}, \texttt{i}, \texttt{se}), \\ \qquad\qquad \texttt{bHeap} := \mathit{store}(\texttt{bHeap}, \texttt{a}, \texttt{i}, \texttt{se})\}\langle {}_{TR}\pi\, \omega\rangle\phi\end{array}}{\Gamma \models \mathcal{U}\langle {}_{TR}\pi\, \texttt{a[i]} = \texttt{se}; \omega\rangle\phi}\ \textsf{arrayAssign}$$

Assuming that arrays `tr` and `ps` are, respectively, transient and persistent, the symbolic execution of this program:

```
tr[0] = ps[0] = 0; #beginTr; tr[0] = 1; ps[0] = 1; #abortTr;
```

results in the following sequence of state updates:

[4] Only arrays can be made persistent or transient in Java Card, regular objects are always persistent. Thus, we only discuss arrays in this context, but our formalisation works for regular objects, too.

$\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{tr}, 0, 0),\ \texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{ps}, 0, 0),$
$\texttt{bHeap} := \texttt{heap},$
$\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{tr}, 0, 1),\ \texttt{bHeap} := \mathit{store}(\texttt{bHeap}, \texttt{tr}, 0, 1),$
$\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{ps}, 0, 1),$
$\texttt{heap} := \texttt{bHeap}$

With these updates, the valuation of $\mathit{select}(\texttt{heap}, \texttt{ps}, 0)$ and $\mathit{select}(\texttt{heap}, \texttt{tr}, 0)$ results in resp. 0 and 1 as required by the Java Card transaction semantics.

Non-atomic Updates The last complication in the transaction rules are the so-called *non-atomic* updates of persistent array elements. Such updates bypass transaction handling, i.e., no roll-back of data updated non-atomically is performed. Updates to transient arrays as defined by Java Card are in fact non-atomic, as they are never rolled back either. We have just introduced a mechanism that prevents the roll-back of transient arrays, by checking the `<transient>` field of the array and providing corresponding state updates. To extend this behaviour to persistent arrays, we allow for the implicit `<transient>` field of an array to be mutable in our logic. In turn, we can temporarily change the assignment semantics for an array by manipulating the `<transient>` field. Concretely, a non-atomic assignment to a persistent array element can be modelled by first setting the `<transient>` field to true, then performing the actual assignment, and then changing the value of `<transient>` back to false. Hence, a non-atomic assignment "`a[i] = se;`" to a persistent array `a`, is simply modelled as:

```
a.<transient> = true; a[i] = se; a.<transient> = false;
```

Then, the array assignment rule we provided above introduces the necessary updates to the regular and backup heaps to achieve transaction bypass.

In Java Card the non-atomic updates are delegated to dedicated API methods, i.e., they are not part of the language syntax. Hence, the manipulation of the `<transient>` field is delegated to the reference implementation of these API methods, and this *emulation* of non-atomic assignments is easily achieved in the actual Java Card programs to be verified by KeY.

Unfortunately, there is one more condition for non-atomic updates that we need to check. A request for a non-atomic update becomes effective only if the persistent array in question has not been already updated atomically (i.e., with a regular assignment) within the same transaction. If such an update has been performed, any subsequent updates to the array are always atomic within the same transaction and rolled back upon transaction abort. We illustrate this with two simple programs operating on a persistent array `a` above on the right, for simplicity we mark non-atomic assignments with `#=` instead of quoting the actual API call that does that. The top program results in `a[0]` equal to 1 (a non-atomic update is in effect), the bottom program rolls `a[0]` back to 0, as the regular assignment "`a[0] = 2;`" disables any subsequent non-atomic assignments, and hence all transaction updates are reverted.

```
a[0] = 0;
#beginTr;
 a[0] #= 1; a[0] = 2;
#abortTr;
```

```
a[0] = 0;
#beginTr;
 a[0] = 2; a[0] #= 1;
#abortTr;
```

To introduce this additional check in the logic, we employ one more implicit field for array objects, `<trUpdated>`, that maintains information about atomic updates. Set to true, it indicates that the array was already updated with a regular assignment, false indicates no such updates and allows for non-atomic updates in the same transaction still to be effective. The new assignment rule for arrays needs to be altered to handle all these conditions and also to record the changes to the `<trUpdated>` field itself. The saturated state updates to be introduced under different conditions in the assignment rule for "`a[i] = se;`" are the following:

Condition	State update
Always	heap := *store*(heap, a, i, se)
!a.<transient>	bHeap := *store*(bHeap, a, <trUpdated>, *TRUE*)
a.<transient> and !a.<trUpdated>	bHeap := *store*(bHeap, a, i, se)

The updates to the `<trUpdated>` field are purposely stored on the backup heap to ease the resetting of this field with each new transaction, because the backup heap is freshly assigned with each new transaction while the regular heap is not. Now, on transaction abort, the heap reverting update filters out any updates to this field on the backup heap using the anonymisation function of the JDL:

$$\texttt{heap} := \mathit{anon}(\texttt{bHeap}, \mathit{allObjects}(\texttt{<trUpdated>}), \texttt{heap})$$

Intuitively, this expresses the operation of copying the contents of heap `bHeap` to `heap`, but retaining the value of the `<trUpdated>` field in all objects in `heap`. This way all manipulations of `<trUpdated>` in proofs are local to a single transaction.

3 Heaps as Parameters in JML*

The previous section spelled out the details of formalising Java Card transactions in JDL. The key point in this formalisation is the modified assignment rule in the sequent calculus that now operates on two heap variables. In some sense, assignment rules are always the core of the program logic – they give semantics of state changes for the verified program. A specification language that describes the program behaviour also deals in a large part with the corresponding state changes (or lack thereof). Hence, one can say there is a special correspondence between the assignment rules and the specification language.

This means that specifications for methods called in transactions should additionally express properties about data on the backup heap together with the framing conditions. To this end we introduced the following extensions. To redirect any object field access `o.f` to a different heap one can use the `\at` operator, e.g., accessing data on the backup heap is expressed with `\at(backupHeap,o.f)`. In this context, the plain field access `o.f` in fact means `\at(heap,o.f)`. Then, for framing specifications, the assignable clauses also take a heap parameter to bind locations with a corresponding heap:

```
assignable[heap] o.f;
assignable[backupHeap] o.g;
```

```
/*@ public normal_behavior
    requires len >= 0 && off >= 0 && off + len <= a.length;
    ensures \result == off + len;
    ensures (\forall int i; i>=0 && i<len; a[off + i] == v);
    requires[backupHeap] JCSystem.getTransactionDepth() == 1;
    requires[backupHeap] a.<transient> ==> !a.<trUpdated>;
    ensures[backupHeap] (\forall int i; i>=0 && i<len;
     \at(backupHeap, a[off + i]) ==
       ((!a.<transient> && \at(backupHeap, a.<trUpdated>) ?
          \old(\at(backupHeap, a[off + i])) : v) );
    assignable[heap,backupHeap] a[off..off+len-1]; @*/
public static int arrayFillNonAtomic(byte[] a, int off, int len, byte v);
```

Fig. 1. Complete JML* specification for one of the Java Card API methods.

Now it is possible to generate separate proof obligations for the framing conditions for the two heaps and correctly apply method contracts in the presence of two heaps.

We generalise this further. Any specification element in JML*, like a precondition specified with the **requires** clause, receives a heap parameter. This parameter specifies the applicability context of the given specification element. In particular, specification elements defined for the backup heap are only considered in verification contexts of an open transaction, i.e., within the marked $\langle_{TR}\cdot\rangle$ modality. Specification elements not annotated with any heap apply to the default heap that is always active. This way we achieve transparency – old style specifications refer to the regular heap by default and retain their previous semantics. An illustration for this is given in Fig. 1, where a complete specification for the Java Card API method for updating chunks of arrays in a non-atomic way is given for both the transaction and non-transaction contexts.

4 Implementation in KeY

Implementing the support for Java Card transactions in KeY was done in two steps. The first step was to generalise the JML* interface to accept multiple heaps and convey the information about them to the proof obligation generation component and the modular reasoning component. This was simply done by considering an arbitrary list of heaps in the corresponding modules rather than referring to the one predefined heap. Until this point the extensions were fully generic, i.e., not specific to the Java Card transaction mechanism in any way. In particular, the generation of concrete formulae for framing conditions remained the same, only now several ones for different heaps are created.

In the second step we added the core formalisation of Java Card transactions to the KeY system. In KeY the logic rules are defined externally, using the so-called taclet language [2, Chap. 4] for defining the corresponding rewrites. The TR marker was added by simply declaring a new modality. Then a handful of

new rules we discussed in Sect. 2 were added to the rule base. As an example, a self-explanatory taclet for the `#beginTr` statement is the following in KeY:

```
beginJavaCardTransaction {
  \find (==> \diamond{.. #beginTr; ...}\endmodality phi)
  \replacewith(==> {backupHeap := heap}
    \diamond_transaction{.. ...}\endmodality phi) };
```

Apart from this rule and the transaction specific rule for array assignments the addition of the second heap variable `backupHeap` required only declaring it. This declaration automatically tells the other components of the KeY system to include it (considering the current verification context) in the corresponding verification tasks, like proof obligation generation or modular application of contracts.

To evaluate our work we revisited our earlier work on the fully verified reference implementation of the Java Card API [19]. We specified the Java Card API methods following the extended JML* syntax (see again Fig. 1) and verified both the reference implementation of the API as well as a handful of other Java Card examples that make calls to the Java Card API.

The overall result of our work shows considerable improvements compared to our old formalisation of transactions [3, 18] back when the heap model in JDL was not based on explicit heap access through a special program variable. The complete set of changes to the logic and the calculus is now much smaller, the implementation overhead of the new rules practically negligible, and finally the resulting automatic proofs for Java Card programs much more readable. We attribute these improvements to the use of multiple heaps, which was not possible before. Previously, the semantics of state updates on the implicit heap had to be heavily modified to include a notion of a *forgetting* update to model data rollback in the logic with deep implications for the calculus and the implementation. Preliminary work in the area of concurrent verification provides another strong case for the explicit use of multiple heaps as we briefly describe next.

5 New Applications for Multiple Heaps in Verification

In the introduction we mentioned distributed computing as an example where multiple heaps should be considered in the computation model, with at least the local and one remote heap. Another scenario is low level reasoning about systems with (possibly multi-level) cache memory, where one heap would represent the cache and one the main memory. Here the verification could concentrate on the data dependencies and synchronisation between the cache and the main memory. Going further, multi-core systems (like GPUs) could be also modelled using multiple explicit heaps, each heap representing the local memory of a single core. Finally, the real-time Java can be also considered in this context, where programs access memories with different physical characteristics on one embedded device [13].

In the context of the ongoing VerCors project[5] [1] we currently concentrate on extending the KeY logic to deal with permission based verification of concurrent programs. Permission accounting is a specification oriented methodology for ensuring race freedom in concurrent programs that allows for efficient thread-local reasoning. Similarly to the implementation of permissions in the Chalice tool [15, 22] we introduce a permission mask to the JDL to keep track of permissions in the verified programs. From our point of view, this permission mask is nothing more than a parallel heap-like structure that stores permission values for each location instead of the actual values. In the first experimental attempt, using the multiple heap framework that we discussed, we simply added a new heap structure to the logic, represented with the program variable `permissions`, to keep track of the permissions that the local Java thread owns. The location assignment and access rules were amended to ensure, respectively, a write or read permission to a given location. Now, using our heap-aware JML*, we can give permission based specifications:

```
requires[permissions] \at(permissions, o.f) == 1;
assignable o.f;
assignable[permissions] o.f;
```

This states that we require a write permission to the location `o.f`, that this location is changed on the actual heap (the regular **assignable**), and also that the permission to the location may be modified, e.g., through permission transfer to another thread. Disregarding any specification clauses associated with permissions, in the example the first and the third line, transforms the specification into a permission unaware specification. This can be useful for verifying permission and functional properties separately. Very basic examples with permissions have been already verified with an experimental version of KeY.

6 Conclusions

In this paper we discussed the use of multiple heaps in formal verification of Java programs using the formalisation of Java Card atomic transactions fully implemented in KeY as an example. We also took the opportunity to give full details of this formalisation that were not yet published elsewhere. In the ongoing work we apply the same methodology to introduce permission based reasoning for concurrent Java programs in KeY. Few other applications in verification have been named as possible directions for more future work.

It seems that none of the other verification systems that we are aware of try to make heap or heap-like structures explicit on the level of the specification language, although certainly some of them indeed use multiple heap or heap-like structures internally. Most notably, the Chalice tool [15, 22] works with two global variables H and P, that, respectively, represent the heap and the permission mask in the Boogie proof obligations. Not exposing the heap in the

[5] http://fmt.cs.utwente.nl/research/projects/VerCors/.

Separation Logic specifications and associated tools [11, 7] seems natural, however, applying them to new verification scenarios named in Sect. 5 becomes significantly more difficult in our opinion.

When it comes to the formalisation of Java Card atomic transactions, only the Krakatoa tool [17] also provides a sound formalisation and implementation of the transaction roll-back that accounts for the specifics of non-atomic methods. The Krakatoa formalisation relies on keeping extra copies of data to be rolled back on the same heap as all the other data in dedicated backup fields associated with regular fields, i.e., all data fields are backed-up separately instead of the whole heap. This is very similar to our first formalisation of transactions [3], which turns out to be very heavy-weight compared to our current work. We believe that our current formalisation can be applied easily in other verification systems, as long as such a system is capable of manipulating the heap variable as we do in the KeY logic. A partial support for Java Card transactions has been also recently reported for the VeriFast platform [10], however, the semantics of the transaction roll-back has not been formalised there. Finally, Java Card transactions have been considered to be formalised in the LOOP tool using program transformation to explicitly model transaction recovery directly in the Java code, but the ideas where never implemented in the tool [9].

Acknowledgements. The work of W. Mostowski is supported by ERC grant 258405 for the VerCors project. We would like to thank Richard Bubel for his insights and invaluable help with the implementation.

References

1. A. Amighi, S. C. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The VerCors project: Setting up basecamp. In *6th Workshop Programming Languages meets Program Verification*, pages 71–82. ACM, 2012.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
3. B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In M. Pezzè, editor, *Fundamental Approaches to Software Engineering*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.
4. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *8th Int'l Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.
5. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.
6. Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, June 2000.
7. D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. *SIGPLAN Notes*, 43:213–226, October 2008.

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 M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004.
10. B. Jacobs, J. Smans, P. Philippaerts, and F. Piessens. The VeriFast program verifier – a tutorial for Java Card developers. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, September 2011.
11. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.
12. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods*, volume 4085 of *LNCS*. Springer, 2006.
13. J. Kwon and A. J. Wellings. Memory management based on method invocation in RTSJ. In *Proceedings, On the Move to Meaningful Internet Systems (OTM)*, volume 3292 of *LNCS*, pages 333–345. Springer, 2004.
14. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
15. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, pages 195–222. Springer, 2009.
16. 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.
17. C. Marché and N. Rousset. Verification of Java Card applets behavior with respect to transactions and card tears. In D. V. Hung and P. Pandya, editors, *4th IEEE Conference on Software Engineering and Formal Methods*. IEEE Press, 2006.
18. W. Mostowski. Formal reasoning about non-atomic Java Card methods in Dynamic Logic. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods*, volume 4085 of *LNCS*, pages 444–459. Springer, 2006.
19. W. Mostowski. Fully verified Java Card API reference implementation. In B. Beckert, editor, *4th Int'l Verification Workshop*, volume 259 of *CEUR WS*, 2007.
20. W. Mostowski and E. Poll. Malicious code on Java Card smartcards: Attacks and countermeasures. In *Smart Card Research and Advanced Application Conference*, volume 5189 of *LNCS*, pages 1–16. Springer, September 2008.
21. P. L. Pallec, A. Saif, O. Briot, M. Bensimon, J. Devisme, and M. Eznack. NFC cardlet development guidelines v2.2. Technical report, Association Française du Sans Contact Mobile, 2012.
22. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. In G. Barthe, editor, *European Symposium on Programming*, volume 6602 of *LNCS*, pages 439–458. Springer, 2011.
23. P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software Conference*, volume 6528 of *LNCS*, pages 138–152. Springer, 2011.
24. K. Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology*, volume 3116 of *LNCS*, 2004.
25. Sun Microsystems, Inc., `http://www.oracle.com`. *Java Card 2.2.2 Runtime Environment Specification*, March 2006.