Chapter 10
Verifying Java Card Programs

Wojciech Mostowski

10.1 Introduction

One of the fundamental assumptions of the KeY project at its beginning was that it should support verification of an actual programming language and handle realistic programs. Back then, complete handling of arbitrary Java programs was still considered an unreachable goal in source code based interactive verification. For that reason a simpler, yet actually existing and officially developed, Java technology was chosen as our verification target, namely Java Card—a considerably stripped down version of Java for programming smart cards [Chen, 2000]. The additional motivation for choosing this particular technology were the corresponding application areas that are subject to strict security requirements. Nowadays, smart cards are widely used in the financial sector (bank cards), telecommunications (SIM cards), identity (electronic passports and identity cards), and transportation (electronic tickets). Many of the products are indeed developed on the Java Card platform, in particular, the Dutch biometric passport is based on the Java Card technology. The security requirements of such applications and, sadly, still often occurring security problems in this area[1] highly justify the use formal verification.

In fact, other verification tools back in the days have chosen the Java Card dialect for similar reasons. The KIV [Balser et al., 2000], Krakatoa [Marché et al., 2004], or Jack [Burdy et al., 2003b] tools all explicitly mention Java Card as a target language in the corresponding publications. Furthermore, Java Card was also in the center of the EU VerifiCard project started in 2001 [Jacobs et al., 2001], that consolidated the efforts to provide verification techniques for Java based smart cards.

Many research teams targeted their effort towards Java Card, but the KeY system was the first tool to implement extensions to support an initially *overseen* by researchers feature of Java Card, the atomic transaction mechanism [JavaCardRTE, Chap. 7]. This *extension* of the Java Card Virtual Machine allows one to group

[1] Two recent high profile cases of security flaws in smart card based applications are the Dutch OV-chipcard [Garcia et al., 2008] and one of the smart card based payment authentication protocol for Internet banking [Blom et al., 2012].

assignments into atomic blocks in the context of two types of writable memory on smart card chips: volatile RAM and persistent EEPROM. From the point of view of a formal verification system, the most complicating factor of the transaction mechanism is the possibility of a programmatic abort of an on-going transaction. The result of such an abort is a program that continues its execution, but with *selected* variable assignments reverted according to the complex Java Card Virtual Machine transaction rules.

Our formalization of Java Card transactions for the previous generation of the KeY system [Beckert and Mostowski, 2003, Mostowski, 2006] was admittedly rather heavyweight with deep changes in the implementation of KeY. Nevertheless, the formalization provided a complete and sound verification framework for Java Card programs. This was illustrated with realistically sized case studies described by Mostowski [2005, 2007].

In this chapter, we discuss a new solution tailored to the explicit heap model of KeY as described in detail in Chapter 3. The core of the new solution is the *simultaneous* use of two heaps during the verification. The first heap is used as usual, and keeps track of the current reachable state of the program for the purpose of evaluating properties to be verified. The second additional heap is used to keep a backup copy of the main heap for the purpose of transaction roll-back in case of an abort. The resulting solution turns out to be very simple and clean, and most importantly, is very modular with respect to the rest of the JavaDL. Further simplifications of our new formalization come from a more pragmatic approach to the notion of a verifiable Java Card program. Following our practical experience with Java Card technology [Mostowski and Poll, 2008], rather than to support every possible and contrived use of transaction related constructs, we opted to support only patterns commonly agreed as safe by security experts [Pallec et al., 2012]. Programs not adhering to these patterns cannot be verified because of the design of the formalization. Furthermore, apart from supporting Java Card, the idea of using more than one heap, in general arbitrary many heaps, provides solutions for other areas of research in Java verification, in particular for permission-based reasoning about concurrent programs.

The chapter starts with introducing the Java Card technology in more detail in the next section, and then continues with explaining the details of our current formalization of transactions based on two heaps in Section 10.3. In Section 10.5 we discuss the (transparent) extensions to JML to accommodate multiple heap references also on the specification level. Section 10.6 discusses the verification of several Java Card example programs. Finally, in Section 10.7 we discuss the applications of reasoning based in multiple heaps in other areas of verification.

10.2 Java Card Technology

In the following, we concentrate on features of Java Card relevant for this chapter. A comprehensive overview of the Java Card technology is described by Chen [2000], and [JavaCardRTE, JavaCardVM] provide a technical reference.

The Java Card technology is now at version 3.0 and divided into two editions: *Classic* and *Connected*. In this chapter, we refer to the classic edition which is still the established standard in the smart card industry and is essentially the same as previous Java Card versions that were only released as one edition. The connected edition is a feature-rich regular Java edition and no longer a subset of Java like the classic edition. However, the connected edition remains still to be fully accepted as it is not widely used by the industry, yet. Also, Java Card devices actually implementing the 3.0 connected version are still scarce, despite the fact that the connected variant of Java Card was proposed more than 8 years ago.

The Java Card technology in general provides means to program smart cards with Java. The classic edition of Java Card consists of a language specification, which defines the subset of permissible Java in the context of smart cards, a Java Card Virtual Machine specification, which defines the semantics of the Java byte-code when run on a smart card, and finally the Java Card API, which provides access to the specific routines usually found on smart cards, in particular the transaction mechanism. What makes Java Card easier for verification is the lack of concurrency, floating-point numbers, or dynamic class loading, and a very small API (less than 100 classes). The important feature that adds complexity to the Java Card environment compared to Java is that programs can 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 at the end of a card session. That is, EEPROM provides storage facilities to the card, and RAM provides computation space. The following are the memory allocation rules:

- all local variables are transient,
- all newly created objects and arrays are by default persistent, and
- when allocated with a dedicated API call, an 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 object reference can be kept in a local variable and be transient itself. A garbage collector is not obligatory in Java Card either. Thus, careless handling of allocated references leads to memory leaks, something that is often addressed in Java Card programming guidelines [Pallec et al., 2012]. Any Java reference, 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 virtual machine that takes appropriate actions according to the memory type associated with a given reference.

Example 10.1. Suppose the following program would be run in a Java Card execution environment:

—— Java Card ——————————————————————————

```
class MyClass {
  int persistentField;
  void method(int parameter) {
    int localVariable = parameter * 2;
    this.persistentField = localVariable;
    if(persistentField < 0) {
      MyClass mc = new MyClass();
      mc.method(persistentField);
    }
  }
}
```

——————————————————————————— Java Card ——

Being an instance field of a class, the value of `persistentField` is stored in a permanent chip memory location. The local variables `parameter` and `localVariable` are stored in the transient RAM memory and are forgotten upon method exit. The assignment `this.persistentField = localVariable;` effectively copies data from one type of memory to another type, but this transfer between different memory types is not explicit in the program. The `mc` variable is also local and hence transient, however, the contents of the freshly created object of class `MyClass` to which `mc` refers is part of the persistent storage, and so are all of its fields. Upon return from `method`, a memory leak occurs—the reference to the freshly created object that was stored only in the local variable is lost. On Java Card devices without garbage collection, such leaked memory is irrecoverable.

Additionally, this particular program would very likely cause a memory overuse problem on an actual device. The continuous allocations of `MyClass` would quickly exhaust the limited amount of available persistent memory (which is usually in the range of 64kB to 128kB), but even sooner the RAM memory (in the range of 4kB) would be filled with the call stack caused by the recursion and cause a stack overflow exception. Hence, it is often advisable not to use recursive method calls on Java Card, and not to do postinitialization object allocation. We come back to the issue of object allocation later in Section 10.3.3.

Objects allocated in EEPROM, like in the example above, provide the only permanent storage to an application. To maintain consistent updates of this persistent data, Java Card offers the atomic transaction mechanism accessible through the API. The following is a brief, but complete summary of the semantics of transactions: Updating a single field or array element is always atomic. Updates can be grouped into transaction blocks, an API call to the static method `JCSystem.beginTransaction()` opens such a block; it 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 the corresponding transaction. The program continues execution with the persistent updates reverted but all the transient updates are still in effect.

Example 10.2. The following program uses a transaction to ensure consistent update of persistent fields a and b:

—— Java Card ——

```
class MyClass {
  int a, b;

  void transferFromAtoB(int num) {
    JCSystem.beginTransaction();
    a = a - num;
    b = b + num;
    if(a < 0 || b < 0) {
      // Too much transfered / overflow
      JCSystem.abortTransaction();
    } else {
      JCSystem.commitTransaction();
    }
  }
}
```

—— Java Card ——

In case that either of the two fields goes beyond its assumed bound (they are supposed to be nonnegative), the transaction is aborted, and the values of both fields are reverted to their respective values when the transaction was started. Regardless of the outcome of the transaction, the program could continue its execution and, e.g., try to update a and b with a smaller value, say num/2. Finally, had the local num variable itself been updated in any way within the transaction, it would maintain its updated value regardless of the transaction outcome.

Finally, the API provides so-called *nonatomic* methods to bypass the transaction mechanism. A nonatomic update of a *persistent* array element is never reverted by an abort, provided the same array was not manipulated with regular assignments earlier in the same transaction. We postpone illustrative examples of nonatomic updates to the next section when we explain the details of the formalization of such updates.

10.3 Java Card Transactions on Explicit Heaps

In the following, driven by examples, we gradually present the complete formalization of the Java Card transaction semantics in JavaDL using two heap variables. To start with, we introduce synthetic transaction statements to the Java syntax. That is, the calculus should allow for symbolic execution of #beginTr, #commitTr, and #abortTr synthetic statements that define the transaction boundaries in the verified program. Bridging the actual API transaction calls from the JCSystem discussed in Section 10.2 to these logic-only statements is a straightforward extension of the verification system. Then, consider the following snapshot of a more realistic Java Card program, where the fields `balance` and `opCount` are persistent, permanently storing the current balance and operation count of some payment application. The local variables `change` and `newBalance` are transient:

—— Java Card ——

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

—— Java Card ——

Following the rules from Chapter 3, symbolic execution of this program piece results in the following sequence of state updates to the heap and local variables—ignoring the transaction statements for now:

$$\begin{array}{l}\{newBalace := 0\}\\ \{heap := store(heap, self, opCount, select_{int}(heap, self, opCount) + 1)\}\\ \{newBalance := select_{int}(heap, self, balance) + change\}\\ \{heap := store(heap, self, balance, newBalance)\}\end{array}$$

The symbolic execution of the `if` statement causes proof splitting, and the last update only appears on the `else` proof branch where `newBalance` is assumed to be nonnegative.

After further simplification, these state updates can be applied to evaluate a postcondition. It could, e.g., query the new value of operation count, i.e., the term $select_{int}(heap, self, opCount)$. The evaluation of this term would indicate a one unit increase with respect to the value stored on the heap before this code is executed.

10.3.1 Basic Transaction Roll-Back

Now let us consider what is required to model the basic semantics of an abort, first under the assumption of a simplified Java Card definition, in which updates to local variables should be kept while updates to persistent locations should be rolled back to the state before the transaction. Up till now, data in these persistent locations are synonymous with the data stored on the heap in the logic. Hence, under the above assumption, it is sufficient to roll back the value of the whole heap. This can be done by introducing two simple symbolic execution rules for transaction statements

`#beginTr` and `#abortTr` that respectively store and restore the value of the heap to and from a temporary heap variable *savedHeap* (the $\langle[\cdot]\rangle$ operator indicates that the rule is applicable to both the box and the diamond modality):

$$\mathsf{beginTransaction}\ \frac{\Longrightarrow \mathcal{U}\{savedHeap := heap\}\langle[\pi\ \omega]\rangle\phi}{\Longrightarrow \mathcal{U}\langle[\pi\ \texttt{\#beginTr;}\ \omega]\rangle\phi}$$

$$\mathsf{abortTransaction}\ \frac{\Longrightarrow \mathcal{U}\{heap := savedHeap\}\langle[\pi\ \omega]\rangle\phi}{\Longrightarrow \mathcal{U}\langle[\pi\ \texttt{\#abortTr;}\ \omega]\rangle\phi}$$

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

$$\begin{aligned}
&\{newBalace := 0\}\\
&\{savedHeap := heap\}\\
&\{heap := store(heap, self, opCount, select_{int}(heap, self, opCount) + 1)\}\\
&\{newBalance := select_{int}(heap, self, balance) + change\}\\
&\{heap := savedHeap\}
\end{aligned}$$

Whatever terms referring to heap contents are to be evaluated with these updates, the result are the values on the heap at the point where it was saved in the *savedHeap* 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 *savedHeap* variable is simply forgotten until a possible subsequent new transaction where *savedHeap* is freshly overwritten with a more recent *heap*.

These rules are sufficient for superficial treatment of transactions in JavaDL under the assumption made at the beginning of this section. Note that, so far, no new or modified assignment rules of any kind were introduced, yet assignments can be canceled.

10.3.2 Transaction Marking and Balancing

The two rules we just introduced do not enforce any order on the transaction statements. Indeed, they allow to successfully verify programs:

`#abortTr; #beginTr;`   or   `#commitTr; #commitTr;`

One solution to enforce balancing of transactions is to keep track of a transaction depth counter and make additional checks upon transaction statements. In fact, this is how the Java Card API methods enforce balancing. In Java Card, transactions cannot be nested, i.e., the maximum allowed depth is always 1. However, by the same specification, an open transaction does not have to be terminated within the same syntactical block of a program, it is only required that the transaction is eventually terminated within the same card session, if not, the transaction is aborted by the system itself. Thus, in principle, a transaction can be initiated in one method, and terminated in another.

In the logic we opt for a more restrictive use of transactions, with the following rationale. Java Card security guidelines [Pallec et al., 2012] ban programs with large transaction blocks spanning over several methods (due to the high risk of overrunning the transaction buffer). A logic calculus allowing for such arbitrary "spreading" of transaction statements would need to carry around complete information about the state of a transaction across method invocations, a complication that would make proofs unnecessarily more cluttered and difficult.

Following this rationale, our formalization not only relies on this transaction use restricted to one method, but also enforces it, in the following way. A transaction marker $_{TR}$ attached to any modality indicates that the current execution context of the verified program is an open transaction. In practice, such a marked modality is simply a separate modality $\langle_{TR}\cdot\rangle$ (or $[_{TR}\cdot]$ as the case may be). Rules for handling transaction opening and closing statements defined only for the adequate modalities automatically enforce correct transaction balancing. Similarly, the absence of a logic rule for an empty transaction modality prevents closing proof goals with open transactions. Overall this forces a complete transaction block to appear in a single verification context, i.e., one method. Furthermore, special array assignment rules (which we introduce shortly) are also defined for the transaction context only, without cluttering any nontransaction context. This keeps regular Java verification efforts clear of any transaction artifacts in the logic without the need to introduce any special Java Card modes or switches in the prover or similar mechanisms.

The new rules for the transaction statements are the following, now with an explicit rule for the commit statement, in which nothing happens to the *heap* variable, but the transaction context is canceled out by changing the transaction modality $\langle\!\![_{TR}\cdot]\!\!\rangle$ back to $\langle\!\![\cdot]\!\!\rangle$:

$$\mathsf{beginTransaction}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathit{savedHeap} := \mathit{heap}\}\langle\!\![_{TR}\pi\,\omega]\!\!\rangle\phi,\Delta}{\Gamma \Longrightarrow \mathcal{U}\langle\!\![\pi\,\texttt{\#beginTr;}\ \omega]\!\!\rangle\phi,\Delta}$$

$$\mathsf{abortTransaction}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathit{heap} := \mathit{savedHeap}\}\langle\!\![\pi\,\omega]\!\!\rangle\phi,\Delta}{\Gamma \Longrightarrow \mathcal{U}\langle\!\![_{TR}\pi\,\texttt{\#abortTr;}\ \omega]\!\!\rangle\phi,\Delta}$$

$$\mathsf{commitTransaction}\ \frac{\Gamma \Longrightarrow \mathcal{U}\langle\!\![\pi\,\omega]\!\!\rangle\phi,\Delta}{\Gamma \Longrightarrow \mathcal{U}\langle\!\![_{TR}\pi\,\texttt{\#commitTr;}\ \omega]\!\!\rangle\phi,\Delta}$$

10.3.3 Object Creation and Deletion

In Section 10.2 we already mentioned that object allocation on Java Card requires special attention due to limited memory of a smart card and absence of a garbage collector. Because of this, according to the Java Card specification, objects created inside transactions require detailed consideration. In short, when aborting a transaction all objects created in that transaction need to be deallocated, regardless of whether these objects are still referenced from objects outside of the transaction. To prevent dangling references, all such references to the deleted objects must be replaced with `null` by the Java Card virtual machine. In other words, the Java Card virtual machine should perform an explicit, *forced* garbage collection upon transaction abort.

This causes problems for our logic, which, being a logic for garbage-collected Java, was never designed to do any object allocation back-tracking (probably, none of the logics for Java are). However, this is also problematic in the actual Java Card virtual machine implementations. It is not uncommon for the implementations to be buggy and lead to serious security issues, as reported by Mostowski and Poll [2008]. Because of that, both security guidelines and newer Java Card specifications strongly discourage object allocation inside transactions altogether.

In our formalization of transactions, we take the same approach, we do not allow objects to be allocated in transaction contexts. This is simply done by *not* providing any object allocation rules for the transaction modalities.

10.3.4 Persistent and Transient Arrays

By default, in Java Card new objects and arrays are allocated in the persistent memory. For scenarios where an object needs to be allocated in transient memory, the Java Card API offers special static methods that redirect allocation to the transient memory, namely:

- `makeTransientBooleanArray(short size, byte transientType)`,
- `makeTransientByteArray(short size, byte transientType)`,
- `makeTransientShortArray(short size, byte transientType)`,
- `makeTransientObjectArray(short size, byte transientType)`.

Hence, only arrays can be allocated in transient memory, contents of other objects are always persistent. When allocating transient arrays, one also specifies, through the `transientType` argument, the moment when the transient memory is cleared. In Java Card the transient memory can be cleared either upon card reset, i.e., when the card session is terminated, or already upon application termination. The latter is useful in practice when, e.g., one application does not wish for any other application to access the data it worked on for security reasons. However, for our formalization, this is irrelevant. But, to be consistent with the API philosophy, in the following we keep using a number to describe the different types of Java Card memory, rather than just a Boolean differentiating between transient and persistent memory types.

In the next step we need to address the issue of separate transaction treatment for persistent and transient arrays in Java Card. Our solution is general enough to also consider the possibility of regular objects to be transient, but we refer only to arrays in our explanations.

So far in our formalization, we roll back the whole contents of the heap. The actual Java Card transaction semantics require that the contents of transient arrays, allocated by the above API methods, are never rolled back. Since in JavaDL 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 simply done by introducing an additional implicit integer field to all objects, called `<transient>`, that maintains the information about the persistency type of an object. Standard JavaDL allocation rules set this field to 0 that denotes persistent objects, while the dedicated API methods for creating transient arrays specify this field to reflect the `transientType` argument discussed above.

Then, when handling assignments, for persistent arrays we take no additional action, for transient arrays we update the value on the heap and additionally update the value on the backup heap *savedHeap*. 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 resulting assignment rule for arrays is the following (for simplicity we skip array bounds checks and similar here, these are no different from the rules discussed in Section 3.6.2):

arrayAssignTransaction

$$\frac{\begin{array}{l}\mathcal{U}\,(a.\texttt{<transient>} \doteq 0) \Longrightarrow \{heap := store(heap, a, i, se)\}\langle\!\![_{TR}\pi\;\omega]\!\!\rangle\phi \\ \mathcal{U}\,(a.\texttt{<transient>} > 0) \Longrightarrow \{savedHeap := store(savedHeap, a, i, se)\} \\ \qquad\qquad\qquad\qquad\qquad\quad \{heap := store(heap, a, i, se)\}\langle\!\![_{TR}\pi\;\omega]\!\!\rangle\phi\end{array}}{\Longrightarrow \mathcal{U}\,\langle\!\![_{TR}\pi\,\texttt{a[i]=se;}\;\omega]\!\!\rangle\phi}$$

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

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

results in the following sequence of state updates:

$$\begin{array}{l}\{heap := store(heap, tr, 0, 0)\}\{heap := store(heap, ps, 0, 0)\} \\ \{savedHeap := heap\} \\ \{heap := store(heap, tr, 0, 1)\}\{savedHeap := store(savedHeap, tr, 0, 1)\} \\ \{heap := store(heap, ps, 0, 1)\} \\ \{heap := savedHeap\}\end{array}$$

With these updates, the evaluation of terms $select_{int}(heap, ps, 0)$ and $select_{int}(heap, tr, 0)$ results in 0 and 1, respectively, as required by the Java Card transaction semantics.

10.3.5 Nonatomic Updates

The last quirk in the semantics of Java Card transactions are the so-called *nonatomic* updates of persistent array elements. Such updates are invoked by dedicated API calls and they bypass transaction handling, i.e., no roll-back of data updated nonatomically is ever performed, similarly to local variables and transient array updates, even though the data is persistent. By this definition updates to transient arrays as defined by Java Card are in fact nonatomic, as they are indeed 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 behavior 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 nonatomic assignment to a persistent array element can be modeled by first setting the `<transient>` field to a positive value, then performing the actual assignment, and then changing the value of `<transient>` back to 0. Hence, a nonatomic assignment `a[i] = se`, where `a` is a persistent array, is simply modeled as:

```
a.<transient> = 1; a[i] = se; a.<transient> = 0;
```

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

Similarly to transient memory allocation, in Java Card the nonatomic updates are delegated to dedicated API methods. Hence, the manipulation of the `<transient>` field is delegated to the reference implementation of these API methods, and this *emulation* of nonatomic assignments is easily achieved in the actual Java Card programs to be verified by KeY.

Unfortunately, there is one more additional condition that Java Card defines for nonatomic updates that we need to check. A request for a nonatomic 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 the following two simple programs operating on a persistent array `a`, for simplicity we mark a nonatomic assignment with `#=`, which would otherwise require a lengthy call to a static API method:

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

The program on the left results in `a[0]` equal to 1 (a nonatomic update is in effect), the program on the right rolls `a[0]` back to 0, as the regular assignment `a[0] = 2;` disables any subsequent nonatomic assignments, and hence all transaction updates are reverted.

To introduce this additional check in the logic, we employ one more implicit field for array objects, `<transactionUpdated>`, 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 nonatomic 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 `<transactionUpdated>` field itself. Without quoting the complete assignment rule again, the complete state updates to be introduced for an assignment `a[i] = se` is the following:

```
{$heap := store(heap, a, i, se)$}
{$savedHeap :=$
     if $select_{int}(heap, a,$ <transient>$) \doteq 0$ then
          $store(savedHeap, a,$ <transactionUpdated>$, TRUE)$
     else
          if $select_{int}(savedHeap, a,$ <transactionUpdated>$) \doteq FALSE$ then
               $store(savedHeap, a, i, se)$
          else $savedHeap$}
```

The updates to the `<transactionUpdated>` field are purposely stored on the backup heap *savedHeap* to ease the resetting of this field with each new transaction. On transaction abort, the heap reverting update filters out any updates to this field on the backup heap using the anonymization function (see Section 3.3.1) of the logic:

$$\{heap := anon(savedHeap, allObjects(\texttt{<transactionUpdated>}), heap)\}$$

This expresses the operation of copying the contents of heap *savedHeap* to *heap*, but retaining the value of the `<transactionUpdated>` field in all objects in *heap*. Thus all manipulations of `<transactionUpdated>` in proofs are local to a single transaction.

10.4 Taclets for the New Rules

The implementation of the new logic rules to handle Java Card transactions that we have just presented is almost trivial. Only a handful of new taclets (see Chapter 4) have to be added to the KeY rule base, most of them very simple. Only the rule for assigning array elements within transactions is considerably more complicated because of the cascade update that needs to be introduced. Yet it only involves changing one kind of rule for array assignments. The only nontaclet extensions, i.e. *internal* to KeY, are:

- the introduction of the additional modalities $\langle\![_{TR}\cdot]\!\rangle$ in the KeY data structures. With the help of schematic modal operators these new modalities are easily added to the existing rules for Java constructs that are not affected by transaction semantics,
- the addition of the built-in Java statements for transaction boundaries `#beginTr`, `#commitTr`, and `#abortTr` to the Java syntax. In the implementation they received fully descriptive names;
- the addition of two new implicit fields to objects, `<transient>` and `<transactionUpdated>`.

Below we give the essential taclets that implement the new rules.

The two additional modalities $\langle_{TR}\cdot\rangle$ and $[_{TR}\cdot]$ are denoted with

```
\diamond_transaction{ ... }\endmodality
```

and

```
\box_transaction{ ... }\endmodality
```

in the taclet language, respectively. The existing rules for all Java constructs are extended to handle these new modalities by including them in the schematic modal operator `#allmodal`:

```
\modalOperator { diamond, box,
    diamond_transaction, box_transaction } #allmodal;
```

Then, the rules for entering and exiting the transactions for the diamond operator are given with the following three taclets:

—— KeY ——————————————————————————

```
beginJavaCardTransactionDiamond {
  \find (==> \<{.. #beginJavaCardTransaction; ...}\> post)
  \replacewith(==> {savedHeap := heap}
      \diamond_transaction{.. ...}\endmodality post)
  \heuristics(simplify_prog)
  \displayname "beginJavaCardTransaction"
};

commitJavaCardTransactionDiamond {
  \find (==> \diamond_transaction{..
               #commitJavaCardTransaction;
             ...}\endmodality post)
  \replacewith(==> \<{.. ...}\> post)
  \heuristics(simplify_prog)
  \displayname "commitJavaCardTransaction"
};

abortJavaCardTransactionDiamond {
  \find (==> \diamond_transaction{..
               #abortJavaCardTransaction;
             ...}\endmodality post)
  \replacewith(==> {heap := anon(savedHeap,
    allObjects(java.lang.Object::<transactionUpdated>),
    heap)} \<{.. ...}\> post)
  \heuristics(simplify_prog)
  \displayname "abortJavaCardTransaction"
};
```
KeY

The rule for the abort also includes the heap update necessary to reset all transaction-updated flags for arrays as described at the end of Section 10.3.5.

Then, the rule for new instance allocations of arrays (see Section 3.6.6.3) needs to be amended to include the initialization of the new implicit fields for transaction handling:

KeY
```
allocateInstanceWithLength {
  \find (==> \modality{#allmodal}{.#pm@#t2()..
                 #lhs = #t.#allocate(#len)@#t;
             ...}\endmodality post)
  \replacewith (==> { heap :=
     store(store(create(heap, #lhs),
       #lhs, java.lang.Object::<transient>, 0),
       #lhs, java.lang.Object::<transactionUpdated>, FALSE) }
     \modality{#allmodal}{..  ...}\endmodality post)
  ...
};
```
KeY

Finally, the rule for assigning array elements is now the following. We skip the null object reference, array index bounds, and array store validity checks for clarity, as they are exactly the same as described in Section 3.6.2:

KeY
```
assignment_to_array_component_transaction {
  \schemaVar \modalOperator { diamond_transaction,
     box_transaction } #transaction;
  \find (\modality{#transaction}{..
           #v[#se] = #se0;
         ...}\endmodality post)
  \sameUpdateLevel
  "Normal␣Execution␣(#v␣!=␣null)": \replacewith(
    {heap := store(heap, #v, arr(#se), #se0)}
    {savedHeap :=
      \if(int::select(heap, #v,
          java.lang.Object::<transient>) = 0)
      \then(store(savedHeap, #v,
          java.lang.Object::<transactionUpdated>, TRUE))
      \else(
          \if(boolean::select(savedHeap, #v,
              java.lang.Object::<transactionUpdated>) = FALSE)
          \then(store(savedHeap, #v, arr(#se), #se0))
          \else(savedHeap)) }
    \modality{#transaction}{.. ...}\endmodality post)
  \add (!(#v=null) & lt(#se, length(#v)) &
      geq(#se,0) & arrayStoreValid(#v, #se0) ==> );
  ... Other proof branches to check for exceptions ...
  \heuristics(simplify_prog, simplify_prog_subset)
};
```

KeY

10.5 Modular Reasoning with Multiple Heaps

The taclets that we have just discussed cover the implementation of the new logic rules to handle Java Card transactions. This makes the core JavaDL calculus and the associated proving engine of KeY aware of transactions. It does not yet, however, handle modular reasoning with contracts in the presence of multiple, simultaneously evolving heap data structures. This issue of modularity for multiple heaps is entirely orthogonal to the transaction semantics. That is, other possible extensions to KeY that involve the use of more than one heap face the same problem. Such additional heaps can be used to model some concrete execution artifact, like the transaction handling described here or different physical memories present on some device. Alternatively, additional heaps can be used to introduce additional abstractions to the logic, for example, permission accounting for thread-local concurrent reasoning. We discuss these possible scenarios in a bit more detail in the concluding Section 10.7.

From the point of view of KeY, heap variables are simply program variables, only that they require special handing when proof obligations are generated and when contract or loop rules are applied (see Chapter 9). In particular, location sets declared in **assignable** clauses are used to properly handle framing. In the presence of multiple, simultaneously changing heaps, the framing conditions, and hence the **assignable** clauses, have to take into account the additional heaps.

To lift this up to the specification layer of JML, we now allow the assignable clauses to take an additional (and optional) argument that declares the heap that the subsequently listed locations refer to. When no heap argument is given, the default memory heap represented with the `heap` variable, is assumed. In this extension of JML, the heap argument is given in angle brackets following immediately the **assignable** keyword. Hence, the following are all valid assignable clauses:

`assignable o.f;`	*refers to the default heap*
`assignable<heap> o.f;`	*the same, but explicit*

`assignable<savedHeap> o.f, o.g;`	*locations that can change on the backup heap*
`assignable<heap><savedHeap> o.f;`	*location that can change on both heaps simultaneously*

For such specifications (for methods and/or loops), KeY can now generate and use appropriate framing conditions over all defined heaps. On the implementation side KeY can handle any arbitrary, but fixed, number of additional heaps. We give more realistic examples of how this is used in practice in the next section.

The idea of making heaps explicit parameters in assignable clauses is further extended to other specification constructs to allow even more flexible reasoning. In principle, not all of the additional heaps are *active* in all verification and/or proof contexts. In particular, `savedHeap` is only relevant in proof contexts where there is an active Java Card transaction, i.e., only within the scope of transaction modalities $\langle\!\![_{TR}\cdot]\!\!\rangle$. In other contexts, all specification expressions relating to the `savedHeap` can be just ignored. Since we allow all of the specification elements, in particular **`requires`** and **`ensures`** clauses, to declare the heaps they relate to, KeY can selectively create method contracts that filter out the correspondingly unused heaps for the different verification and proof contexts. Conversely, the presence of additional heaps in the specification give the KeY system an indication whether the given contract is applicable in a given context. In particular, contracts that are not listing assignable locations for the `savedHeap` are not considered applicable in transaction contexts.

Apart from assignable and other specification clauses, the user has to be able to refer to locations on different heaps in regular JML expressions. To this end, the operator **`\backup`** can be applied to any field or array element access expression to indicate that the backup heap `savedHeap` should be accessed rather than the regular one, exactly in the same way as the **`\old`** operator accesses the heap prior to the method call. Other additional heaps would introduce their own corresponding access keywords, for example, **`\perm`** to specify concurrent access permissions stored a separate heap (again, see the end of the concluding Section 10.7).

The final extension for JML to fully access the Java Card transaction extensions is the ability to access the implicit fields `<transient>` and `<transactionUp-dated>` from specifications. Both can be simply accessed directly as field references. Additionally, for any object `o`, the former can be access with a proper Java Card API call, `JCSystem.isTransient(o)`, the latter can be accessed with the JML keyword **`\transactionUpdated`**`(o)`.

10.6 Java Card Verification Samples

This section is devoted to a handful of examples that make use of the presented formalization of Java Card transactions. We start with the JML specified version of the balance updating method that we used as a running example in Section 10.3, then we discuss how the actual reference implementation of the native transaction-*sensitive* array manipulation routines of the Java Card API can be specified and verified with KeY. Finally, we show a short example of Java Card code implementing a pin counter update routine using one of these API methods. It illustrates the verifiability of the effects of nonatomic updates to ensure the routine's security. All of the presented examples are verified with KeY fully automatically, hence we discuss mostly the specifications.

10.6.1 Conditional Balance Updating

Below we list the `updateBalance` example we used earlier in Section 10.3 with full JML specifications. The method has two specification cases to reflect the two possible outcomes of the method: either both object fields are updated or none of them is. Additionally, we specify that both fields should never be negative:

—— Java + JML ——

```
public class PurseApplet {

  short balance = 0; //@ invariant balance >= 0;
  short operationCount = 0; //@ invariant operationCount >= 0;

  /*@ public normal_behavior
        requires JCSystem.getTransactionDepth() == 0;
        requires balance + change >= 0;
        ensures balance == \old(balance) + change;
        ensures operationCount == \old(operationCount) + 1;
        ensures \result == \old(balance) + change;
      also
      public normal_behavior
        requires JCSystem.getTransactionDepth() == 0;
        requires balance + change < 0;
        ensures balance == \old(balance);
        ensures operationCount == \old(operationCount);
        ensures \result == \old(balance) + change;
    @*/
  public short updateBalance(short change) {
    short newBalance = 0;
    JCSystem.beginTransaction();
    this.operationCount++;
    newBalance = (short)(this.balance + change);
    if(newBalance < 0) {
      JCSystem.abortTransaction();
      return newBalance;
    }
    this.balance = newBalance;
    JCSystem.commitTransaction();
    return newBalance;
  }
  ...
}
```

——— Java + JML ———

The newly calculated balance is returned by the method, regardless of the actual outcome of the balance updating, i.e., the method also returns a possibly negative would-be balance. This is to illustrate that updates to local variables are not at all affected by transactions. In both specification cases, the method's result is the same, i.e., equal to the sum of the initial balance and the requested change. Also, both specification cases require that there is not any on-going transaction when this method is called. That is, the transaction depth recorded by the API should be 0. The remaining parts of the two specification cases spell out the property that the two object fields are updated simultaneously. If the new requested balance is not negative, the operation count is increased and the new balance is stored. Otherwise, both fields remain unchanged.

10.6.2 Reference Implementation of a Library Method

As mentioned in Section 10.2, the Java Card API is a substantially cut-down version of the regular Java API. The classic edition Java Card API consists of less than 100 classes, moreover, a lot of the methods in the API are not implemented in Java Card themselves, but as native code. This is because most of the API is an interface to the smart card hardware: an Application Protocol Data Unit (APDU) buffer that is used for communication with the host, the transaction facilities that we have been discussing here, or the cryptographic facilities usually supported by a dedicated CPU.

One particular part of this API is responsible for efficient handling of bulk memory updates, i.e., complete array updating or copying. These *native* array methods not only improve the performance, but also offer transaction specific handling of the arrays. That is, one class of array methods ensure atomic update of the given array (as if the whole operation were included in a transaction block), the second class allow for nonatomic updates of array elements. Such nonatomic updates, as discussed in Section 10.3.5, allow by-passing an on-going transaction to effectively enforce unconditional update of persistent memory that would otherwise be reverted by a transaction abort.

From the point of view of the reference implementation, these nonatomic methods are the most interesting and the most challenging. Here we discuss one that fills an array with data, the `arrayFillNonAtomic` method from the `Util` class with the following signature:

```
public static final short arrayFillNonAtomic(
    byte[] bArray, short bOffset, short length, byte value)
  throws NullPointerException, ArrayIndexOutOfBoundsException;
```

In regular execution contexts, i.e., outside of any transaction, this method does what one would expect: It fills the range of array elements from `bOffset` to `bOffset+length-1` incl. with the `value` byte. The return value is the first offset index right after the modified elements. The specification is straightforward with JML, our specification also requires conditions for normal termination:

—— Java + JML ——

```
public normal_behavior
  requires JCSystem.getTransactionDepth() == 0;
  requires bArray != null && length >= 0;
  requires bOffset >= 0 && bOffset+length <= bArray.length;
  ensures (\forall short i; i >= 0 && i < length;
             bArray[bOffset+i] == value);
  ensures \result == bOffset + length;
  assignable bArray[bOffset..bOffset+length-1];
```

—— Java + JML ——

The first precondition limits the use of the method to nontransaction contexts only. The code to achieve this behavior is also straightforward, this is done with a simple loop specified with JML:

—— Java + JML ——

```
/*@ loop_invariant i >= 0 && i <= length &&
      (\forall short j; j>=0 && j<length;
         bArray[bOffset + j] == (j < i ?
             value : \old(bArray[bOffset + j]))
      );
    decreases length - i;
    assignable bArray[bOffset..bOffset+length-1]; @*/
for(short i=0; i<length; i++) {
  bArray[bOffset + i] = value;
}
return (short)(bOffset + length);
```

—— Java + JML ——

The verification of this poses KeY no problems whatsoever.

Extending both the specification and the reference implementation to the transactional behavior makes things a little bit more complicated. First of, our top-level specification needs to state what the effects on the contents of the `savedHeap` are going to be. This is specified with the following and reflects the semantics of nonatomic updates described in Section 10.3.5 above.

—— JML ——

```
public normal_behavior
```

```
  requires JCSystem.getTransactionDepth() == 1;
  requires !\transactionUpdated(bArray);
  requires JCSystem.isTransient(bArray) ==
      JCSystem.NOT_A_TRANSIENT_OBJECT;
  requires bArray != null && length >= 0;
  requires bOffset >= 0 && bOffset+length <= bArray.length;
  ensures (\forall short i; i >= 0 && i < length;
      bArray[bOffset+i] == value);
  ensures \result == bOffset + length;
  ensures (\forall short i; i>=0 && i<length;
      \backup(bArray[bOffset + i]) == value);
  assignable<heap><savedHeap>
      bArray[bOffset..bOffset+length-1];
```
JML

The presence of the savedHeap reference in the assignable clause makes this specification applicable to transaction contexts, the first precondition narrows this to transaction contexts only. Then, we limit ourselves only to persistent arrays and require that the array has not been updated with regular assignments in the same transaction, i.e., to execution contexts where the update caused by the method is indeed nonatomic. The remaining preconditions are as before to ensure nonexceptional behavior of the method. The first two postconditions are the same as before and specify the effects of the method on the state of the regular heap. The third postcondition states that the contents of bArray is also changed on the backup heap. Effectively this means that the method bypasses the transaction and updates the array unconditionally.

The implementation of the method and the specification of the loop also need to be changed accordingly. First, to emulate the nonatomic update in the code we need to temporarily change the persistency type of the array, following the schema described in Section 10.3.5 above. This should only be done in case the array is not already transient. Thus, we surround the loop with the following:

Java Card

```
final boolean changeTransient = (JCSystem.isTransient(bArray)
    == JCSystem.NOT_A_TRANSIENT_OBJECT);
if(changeTransient) {
    JCSystem.nativeKeYSetTransient(bArray,
        JCSystem.CLEAR_ON_RESET);
}

// The update loop...

if(changeTransient) {
    JCSystem.nativeKeYSetTransient(bArray,
        JCSystem.NOT_A_TRANSIENT_OBJECT);
}
```
——— Java Card ———

The method `nativeKeYSetTransient` is a built-in KeY method that can change the `<transient>` field of objects, something that is not normally possible with regular Java syntax.[2]

Finally, we add specifications to our update loop to also account for changes on the `savedHeap`. The loop invariant quoted before stays the same, as does the **`decreases`** clause. We add a new loop invariant that states the effects of the loop on the backup heap and adds this heap to the assignable clause:

——— JML ———

```
// Previous loop invariant and decreases clause
loop_invariant (\forall short j; j >= 0 && j < length;
    \backup(bArray[bOffset + j]) == value);
assignable<heap><savedHeap>
    bArray[bOffset..bOffset+length-1];
```

——— JML ———

KeY also has no problems verifying this modified method automatically. The final remark for this method is that it is clear that we introduced redundancy in the specification. The first, nontransactional specification is practically a subset of the second specification. This was done purposely for the clarity of the presentation. However, it is possible to combine the two specifications (both the method contract and the loop specification) into one and make KeY construct a contract that is applicable in both nontransactional and transactional contexts. This is done by also annotating the **`requires`** and **`ensures`** clauses with the corresponding heap variable. Then, in a nontransaction context only the clauses *not* annotated with `savedHeap` are used, in transaction contexts both clauses are used. In particular, the postconditions that specify quantifier expressing the new state of the array would now be specified in one contract with:

——— JML ———

```
ensures (\forall short i; i >= 0 && i < length;
    bArray[bOffset+i] == value);
ensures<savedHeap> (\forall short i; i >= 0 && i < length;
    \backup(bArray[bOffset+i]) == value);
```

——— JML ———

[2] One can consider the `<transient>` field purposely hidden, so that only the API is allowed to make changes of this field that are legal in terms of Java Card specification.

10.6.3 Transaction Resistant PIN Try Counter

We can now use the method `arrayFillNonAtomic` to implement a transaction-safe PIN (Personal Identification Number) try counter [Hubbers et al., 2006]. A try counter is a simple one variable counter that gets decreased with every authentication event, i.e., entering the PIN. Only when the entered PIN is correct, the try counter gets reset to its predefined maximum value (typically 3). Once the counter reaches zero, no further authentication attempts should be possible, and resetting the PIN back to a usable state requires some sort of a administrative procedure, for example, entering a PUK (Personal Unlocking Key) code or simply getting a new card.

This describes the security of the PIN try counter on the functional level, and up to this point such a try counter is trivial to implement. On top of this functionality, one needs to ensure that the try counter is resistant to all sorts of attacks. Using the transaction mechanism, one of the possible attacks on a PIN is the following. The try counter of the PIN is stored in the persistent memory of the card so that the PIN state is not reset during every new card session. Then, one could use an aborting transaction to revert the try counter. The PIN checking routine is enclosed in a transaction. The attacker guesses a PIN code and in case the PIN is not correct the transaction is aborted, which in effect reverts all the updates to the persistent memory, and that would include the try counter. Effectively this gives the attacker an infinite number of possible guesses to break the PIN code. And, given the usually short PIN numbers (4 or 6 decimal digits), this breaks the security of the card wide open. Such an attack is realistically possible if an attacker is able to upload such an exploiting applet that would attempt to break the global PIN of the card, normally accessible through one of the security APIs of Java Card.

To tackle this problem, the try counter needs to be updated in a nonatomic fashion, so that no transaction aborts would ever affect its value. Although the Java Card API provides nonatomic updates for arrays only, it is still possible to implement a try counter using these methods. Instead of storing the counter in a single variable (object field), we store the counter in a one element array. Then we simply always use a nonatomic method from the API to update the try counter.[3] For this we provide the following Java Card class:

—— Java Card ——————————————————————

```
final public class TryCounter {
  private byte[] counter = new byte[1];
  private final byte max;

  public TryCounter(byte max) {
    this.max = max;
    reset();
  }
```

[3] To prevent denial-of-service attacks, also resetting the counter to its maximum value should be done in a nonatomic ways. Otherwise, a correct PIN may not be able to reset the counter properly when any possible on-going transaction gets aborted or interrupted.

```
  public void reset() {
    Util.arrayFillNonAtomic(counter, (short)0, (short)1, max);
  }

  public boolean decrease() {
    if(counter[0] == (byte)0) return false;
    byte nv = (byte)(counter[0] - 1);
    Util.arrayFillNonAtomic(counter, (short)0, (short)1, nv);
    return true;
  }

  public byte get() {
    return counter[0];
  }
}
```

——————————————————— Java Card ——

For verification we shall also provide a handful of JML annotations. We concentrate here on the decrease method. The invariants that we need are the following:

—— JML ———————————————————

```
invariant counter != null && counter.length == 1 &&
  JCSystem.isTransient(counter) ==
    JCSystem.NOT_A_TRANSIENT_OBJECT;
invariant !\transactionUpdated(counter) && counter[0] >= 0;
```

——————————————————— JML ——

Then the contract for decrease in most part reflects the contract for arrayFill-NonAtomic as decrease essentially just calls arrayFillNonAtomic:

—— JML ———————————————————

```
public normal_behavior
  ensures \result <==> (\old(counter[0]) != 0);
  ensures counter[0] == \old(counter[0]) - (\result ? 1 : 0);
  ensures<savedHeap> \backup(counter[0]) ==
    \old(counter[0]) - (\result ? 1 : 0);
  assignable<heap><savedHeap> counter[0];
```

——————————————————— JML ——

We are now ready to specify and verify our security property, i.e., that the counter is decreased regardless of any on-going (and aborting) transaction. We do this by writing an auxiliary test method with the following specification:

—— Java + JML ———————————————————

```
/*@ normal_behavior
     requires \invariant_for(c) && c.get() > 0;
     ensures \invariant_for(c) && c.get() == \old(c.get()) - 1;
  @*/
void testCounter(TryCounter c, boolean abort) {
  JCSystem.beginTransaction();
  c.decrease();
  if(abort) {
    JCSystem.abortTransaction();
  } else {
    JCSystem.commitTransaction();
  }
}
```

Java + JML

The unspecified Boolean parameter `abort` makes KeY consider both transaction cases and effectively verifies that the counter is always decreased. Similar specifications and additions to the `TryCounter`'s invariant are required to also verify the `reset()` method's resistance to transactions, we leave this is as an exercise for the reader.

10.7 Summary and Discussion

This chapter discussed a complete solution to reason about Java Card transactions in JavaDL using KeY. The base of the solution is the manipulation of two memory heaps to model the effects of selective transaction roll-backs. This formalization is fully implemented in KeY, and we presented complete relevant verification examples of realistic Java Card programs.

In its previous instance [Beckert et al., 2007], the KeY system was the first verification system to fully handle Java Card platform intricacies. The current formalization presented in this chapter improves considerably over the previous one discussed in detail in [Beckert and Mostowski, 2003] and [Mostowski, 2006], while considerably sized Java Card verification case studies done with the previous version of KeY are discussed in [Mostowski, 2005] and [Mostowski, 2007]. In the new formalization, there is no need to introduce new semantics for state updates of KeY and the number of additional (transaction marked) modalities is smaller. Also the number of logic rules that extend the basic KeY logic is very low compared to our previous work. As for efficiency, there is not much to discuss—all the examples from this chapter verify fully automatically with KeY in a matter of seconds. The simplicity of our current solution also underlines the good choice of modeling the memory with explicit heap variables—explicit manipulation of these variables makes our Java Card reasoning model very compact.

10.7.1 Related Work

We are only aware of one more verification system that formalized Java Card transactions with similar success to KeY, namely the Krakatoa tool [Marché and Rousset, 2006]. There, a complete solution is also provided with corresponding extensions to JML and implemented in the Why verification platform. When it comes to simplicity and reusability of their solution, we would place it somewhere in between our old and our new solution. In particular, the notion of backup heap locations is used there, however, each single memory location has its own corresponding backup cell on the same heap as the original memory locations, instead of the whole heap being copied/mirrored like in our solution. A support for Java Card transactions has been also reported for the VeriFast platform [Jacobs et al., 2011a,b], however, it is not clear how and if the semantics of the transactions has been formalized there. Finally, Java Card transactions have been considered to be formalized 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 [Hubbers and Poll, 2004].

10.7.2 On-going and Future Research

The use of multiple heaps is a generic feature of our formalization. That is, the same methodology of simultaneous manipulation of several heaps can be used to model other specific features of (Java) programs or to add new elements to the verification model. In particular, in distributed systems one can consider the local and remote memories as separate heaps. Similarly, multi-level caches can be treated by assigning separate heap to each cache. Going further, multi-core systems (like GPUs) could be also modeled using multiple explicit heaps, each heap representing the local memory of a single core. Finally, real-time Java can be also considered in this context, where programs access memories with different physical characteristics on one embedded device.[4]

10.7.3 Multiple Heaps for Concurrent Reasoning

In the context of the VerCors project[5] [Amighi et al., 2012], we concentrate on extending the KeY logic with permission accounting to enable thread-local verification of concurrent Java programs. The VerCors project is concerned with the verification of concurrent data structures, both from the point of view of possible thread interfer-

[4] Admittedly, in the last two examples the different memories are likely to be disjoint and not sharing any heap locations, hence not really utilizing the full power of reasoning with multiple heaps.

[5] www.utwente.nl/vercors/.

ence and with respect to meaningful functional properties. Permission accounting [Boyland, 2003] is a specification oriented methodology for ensuring noninterference of threads. Single threads verified with respect to permission annotations (specifying rights to read and/or write memory locations) are guaranteed to be data-race free. This approach is very popular in verification methods based on Separation Logic or similar concepts [Reynolds, 2002]. In particular, the VeriFast [Jacobs et al., 2011b] system implements fractional permission accounting, and the Chalice [Leino et al., 2009] verifier also uses permission annotations to verify concurrent programs specified with implicit dynamic frames style specifications [Smans et al., 2012]. The VerCors project also employs permission accounting in its own version of Separation Logic [Amighi et al., 2014b] and its corresponding automated tool set [Blom and Huisman, 2014]. Adapting KeY to incorporate permission accounting is a step in providing also interactive verification support in the VerCors project [Huisman and Mostowski, 2015, Mostowski, 2015].

The essential part of adding permission accounting to KeY is the addition of the parallel permission heap to the logic, simply represented with the `permissions` heap variable. This heap stores access permissions to object locations while the corresponding location values are stored on the regular heap like before.[6] Each access to the regular heap is guarded by checking the corresponding access right on the permission heap. Locations on the permission heap can also change like on the regular heap. Namely, permissions to single object locations are mutated when permission transfer occurs in the verified program, that is, upon new object creation, forking and joining of threads, and acquiring/releasing synchronization locks (either through the **`synchronized`** blocks or through dedicated Java concurrency API classes). As with the Java Card transaction treatment the `permissions` heap is explicit in the specifications. In particular, locations listed in the assignable clause for permission indicate possible permission transfer. Conversely, locations not listed in such an assignable clause are guaranteed to preserve their permissions.

Example 10.3. At the time of the writing of this chapter, the current official version of KeY already implements permission accounting as an experimental feature. For example, the three following (admittedly artificial) methods annotated with permission specifications are easily verifiable with KeY:

—— Java + JML ——————————————————

```
public class MyClass {

  int a, b;
  Object o;

  /*@ public normal_behavior
       requires<permissions> \writePerm(\perm(this.o));
```

[6] In fact, this is the same way of treating permissions as in the Chalice verifier [Leino et al., 2009]. In Chalice memory locations are stored on the regular heap denoted with H, while permissions reside in the permission mask denoted with P. In Chalice, however, these variables are hidden from the user in the depth of the module translating the specifications and programs to the SMT solver.

```
      ensures<permissions>
        \perm(this.o) == \old(\perm(this.o));
      ensures \fresh(this.o);
      assignable<heap><permissions> this.o;
    @*/
  public void method1() {
    o = new Object();
  }

  /*@ public normal_behavior
      requires<permissions> \writePerm(\perm(this.o));
      ensures \fresh(this.o);
      assignable<heap> this.o;
      assignable<permissions> \nothing;
    @*/
  public void method2() {
    o = new Object();
  }

  /*@ public normal_behavior
      requires<permissions>
        \writePerm(\perm(this.a)) && \readPerm(\perm(this.b));
      ensures this.a == this.b;
      assignable this.a;
      assignable<permissions> \nothing; @*/
  public MyClass method3() {
    this.a = this.b;
    MyClass tmp = new MyClass();
    tmp.a = 1;
    tmp.b = 1;
    return tmp;
  }
}
```

Java + JML

The references to the `permissions` heap in the specification clauses follow the same pattern as described in Section 10.5 in the context of the `savedHeap` variable. The additional JML keywords present in the specifications are **`\perm`**, **`\readPerm`**, and **`\writePerm`**. The first one is analogous to the **`\backup`** operator, and it redirects object location look-up to the permission heap instead of the regular heap. The other two keywords are operators to interface JML with the corresponding predicates in the logic that evaluate the underlying permission expressions to establish the resulting access rights. Permission expressions as such are an orthogonal issue to heap handling. Many verification systems use fractional permissions in the range $(0,1]$ represented with rational numbers [Boyland, 2003], where 1 denotes a full access (write) permission, and any positive number strictly less than 1 a partial access (read) permission. In KeY, we are using fully symbolic permission expressions that are described in detail in [Huisman and Mostowski, 2015]. However, in the specification that we present here the permission values are abstracted away to be simply read or write permissions, and they could be anything, including the classical fractional permissions.

The first two methods are identical in code, but they differ in the specifications. For `method1()` we state that the permission to `this.o` might be changed by this method, and in the postcondition we specify what this change is going to be, simply that the permission stays the same. Thus, the specification can be optimized to what is stated for `method2()`, i.e., that the permission to `this.o` does not change at all. For `method3()` we need to require that there is a read permission to `this.b` and a write permission to `this.a`. This ensures that the first assignment in the method is valid. Then a new object is created and both its fields are assigned with new values. None of these remaining statements require any additional access permissions. This is because local variables are always fully accessible, and new objects are always allocated with full access permission to the current thread. As in the other two methods, apart from creating new objects, `method3()` does not do any permission transfers, hence the assignable `\nothing` for the permission heap. Note that because of the object creation, we cannot specify `\strictly_nothing` for any of these methods as this would exclude new object creation, which also affects the permission heap.

At the time of writing this, the full support for permission accounting is not yet finished in the KeY system and should be considered experimental. The crucial missing element is the verification of specifications themselves. That is, in permission-based concurrent reasoning, it is only sound to express properties about object locations to which the specification has at least a read access. That is, similarly to the accessible condition checking for self-framed footprints described in Section 9.3, all permission-based specifications need to frame themselves in terms of read access permissions. The details of the self-framing of permission-based specification in KeY are described in [Mostowski, 2015], where we also discuss modular specifications for API synchronization classes using JML model methods (see Section 8.2.3).