Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D3.7

Report on multithreading

Due date of deliverable: 2009-03-01(T0+42)

Actual submission date:

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **INRIA**

Final version

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	**Contributed to Chapter**
INRIA	1, 2, 3, 4, 5
RUN	1, 3, 5
UEDIN	1, 2, 5

This document was written by David Aspinall (UEDIN), Christian Haack (RUN), Marieke Huisman (INRIA), Clément Hurlin (INRIA), Gustavo Petri (INRIA), Erik Poll (RUN) and Jaroslav Ševčìk (UEDIN).

Executive Summary:
Report on multithreading

This document summarizes deliverable D3.7 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including this deliverable, is available on-line at `http://mobius.inria.fr`.

This document describes the final results of Task 3.3 on the verification of multithreaded applications. We describe the semantics of multithreaded Java (considering both the Java Memory Model, and the behavior at bytecode level, including relevant instructions and native methods), and we describe a program logic, based on separation logic with permissions. Finally, we show how the program logic also can be used to show that a program adheres to a certain usage protocol, and to derive provably correct program transformations. This summarizes the research published in [3, 22, 23, 24, 25, 26, 27, 30, 31, 32, 33, 15, 16].

The document is structured as follows. Chapter 2 describes how a formal model of the Java Memory Model can be used to derive whether standard program transformations are behavior preserving. It also presents a complete formalization of the multithreaded Java Virtual Machine, including monitor-related instructions, relevant native methods (such as thread creation, and the wait-notify mechanism), exceptions and the interrupt mechanism. For any data race free program, its behavior can be described with this formalization.

Chapter 3 describes a variant of separation logic with permissions, that handles all relevant features of multithreaded Java, i.e., it deals with object-orientation, the fork-join mechanism and reentrant locks. Because of the combination with abstract predicates, the permission system gives an elegant and expressive specification and verification system. This is illustrated on the (concurrent) Iterator pattern.

Finally, Chapter 4 shows two derived uses of the program logic. First it shows how it can be used to specify usage protocols, that describe in what order methods may be called. Second it shows how a program that is proven correct can be parallelized and optimized, together with its correctness proof. The structure of the correctness proof is used to decide which optimizations can be applied.

Contents

Chapter 1

Introduction

Multithreading is one of the major challenges in verifying security requirements on applications in global computing scenarios. In the last decade, several logics (and tools) have been developed to reason about single-threaded programs [29, 35, 50]. For multithreaded applications, initial theoretical investigations have been made to develop a logic-based verification method [1, 2, 49, 44], but so far this has not led to a practical, tool-supported method, with the possible exception of [34], as the most mature approach to date.

This report describes the outcome of our investigations on how to develop a practical and sound verification technique for multithreaded applications. To achieve this goal, we concentrated on two different aspects: the semantics of multithreaded Java and the development of a practical verification method.

Our investigations concerning the semantics of multithreaded Java are described in Chapter 2. First, we have further investigated the Java Memory Model [39], which describes all legal executions of a multithreaded application. The intermediate deliverable described a formalization of the Java Memory Model, this report presents how it can be proven that program transformations are behavior preserving. We also show that some common program transformations are not safe. An important feature of the Java Memory Model is that it guarantees that programs without race conditions can be described by an interleaving semantics. Therefore, in the intermediate report on multithreading, we presented a revival of the RCC race condition checker and on-going work on the formalization of an interleaving semantics for multithreaded Java bytecode, called BicolanoMT. This report presents the completion of BicolanoMT. Compared to the initial presentation, exceptions and the interrupt mechanism have been added. In addition, the extension mechanism of Czarnik and Schubert [14] has been used to reuse the sequential Bicolano formalization as much as possible. This sequential Bicolano semantics was developed in Task 3.1 (see in Deliverable 3.1 [40]) and the formalisation is available on-line at `http://mobius.inria.fr/twiki/bin/view/Bicolano`.

Concerning the development of a practical verification method, described in Chapter 3, our investigations have resulted in a program logic, based on separation logic with permissions [11, 10], for a multithreaded object-oriented language with a fork-join mechanism (i.e., dynamic thread creation and termination) and reentrant locks, thus supporting the main characteristics of multithreaded Java. An important feature of our specification language is the support for abstract predicates, which results in a flexible and expressive specification mechanism. To illustrate the approach, it has been applied to the commonly used (concurrent) Iterator pattern.

Finally, the development of the program logic also gave rise to different derived methods that help to construct correct programs. Chapter 4 gives two examples. First, we show how the program logic can be used to specify and check adherence to usage protocols, that describe the order in which different methods in a class must be called. Second, we present an approach to parallelize and optimize programs in a provably correct way. This technique requires a correctness proof for a (sequential) program, and it uses the structure of the proof to transform the program and its correctness proof. Program transformations that are supported by this method are not only parallelization, but also for example early disposal and late allocation of objects, and early releasing and late acquiring of locks.

Compared to the intermediate report on multithreading, we have not further investigated explicitly the notions of contract-atomicity, contract-independence, thread ownership and several other proposed keywords, as the need for this was subsumed by the development of the specification language with permissions and abstract predicates, which allows to express these concepts in an elegant and uniform way.

This report summarizes the following papers:

- Section 2.1 summarizes results on the validity of program transformations in the Java Memory Model from [16] and Jaroslav Ševčík's PhD thesis [15], which buids on earlier work [3, 30] in this task;
- Section 2.2 summarizes [31], which describes BicolanoMT;
- Section 3.1 summarizes [25, 26], which describe the program logic for the fork-join mechanism;
- Section 3.2 summarizes [22, 23], which describe the program logic for reentrant locks;
- Section 3.3 summarizes [24, 27], which describe the use of the program logic for resource usage protocols, using the Iterator pattern as leading example;
- Section 4.1 summarizes [33], which discusses the use of the program logic to specify and check adherence to protocols;
- Section 4.2 summarizes [32], which proposes a technique to obtain provably correct program transformations using proof trees.

In addition to the papers mentioned above, the work in chapters 3 and 4 will also be the subject of Clément Hurlin's forthcoming PhD thesis.

Chapter 2

Semantics of Multithreaded Java

We distinguish two different topics that are relevant for the description of the semantics of multithreaded Java. The first is the Java Memory Model [39], which describes all legal behaviors of a Java program. In the earlier, intermediate deliverable, we described a formalization of the Java Memory Model's guarantee for data race free programs. Below, we investigate legality of common program transformations in the Java Memory Model (JMM). That is, we address the question whether program optimizers implement the Java Memory Model. Second, we study the behavior of race-free programs. This behavior can be captured by an interleaving semantics. For program verification, we assume that programs are (proven to be) race-free – the behavior of programs with race conditions is too complex to prove their correctness. To provide a formal basis, we extend the Bicolano formalization of the Java Virtual Machine [46, 40] with interleaving, typical multithreaded instructions (`monitorenter` and `monitorexit`), relevant native methods (e.g., `Thread.start()`, `Thread.join()`, `Thread.wait()` and `Thread.notify()`), exceptions and the interrupt mechanism.

2.1 Validity of Program Transformations in the Java Memory Model

In our earlier work, we have verified that the Java Memory Model guarantees interleaving semantics for data race free programs [30, 3], the so-called DRF guarantee. This guarantee allows us to reuse standard reasoning based on interleaving semantics for data race free programs. The other goal of the Java Memory Model was to enable as many program transformations as possible. In this deliverable we describe our investigation of validity of standard thread-local transformations.

We summarise our results in Table 2.1. We have a formal proof for each legal transformation and a counterexample for each illegal transformation [15]. The table shows validity of transformations under three memory models:

SC stands for sequential consistency, i.e., the interleaving semantics.

DRF guarantee column shows which transformations maintain interleaving semantics for data race free programs. The question mark means that we have neither a proof of validity nor a counterexample. We conjecture that the irrelevant read introduction is valid.

JMM column shows validity in the Java Memory Model.

We only consider transformations on adjacent statements, although the underlying trace-semantic techniques can be used to reason about more complex transformations, such as hoisting reads/writes from a loop. Below, we describe the transformations using simple examples.

2.1.1 Transformations by Example

In the examples, `x` and `y` stand for shared memory locations, `r1`, `r2` for thread-local locations, and `m` for a synchronisation monitor.

Table 2.1: Validity of transformations in the DRF guarantee and in the JMM.

Transformation	SC	DRF guarantee	JMM
Trace-preserving transformations	✓	✓	✓
Reordering normal memory accesses	×	✓	×
Redundant read after read elimination	✓	✓	×
Redundant read after write elimination	✓	✓	✓
Irrelevant read elimination	✓	✓	✓
Irrelevant read introduction	✓	?	×
Redundant write before write elimination	✓	✓	✓
Redundant write after read elimination	✓	✓	×
Roach-motel reordering	×(✓for locks)	✓	×
External action reordering	×	✓	×

Trace-preserving transformations do not alter any trace of shared memory operations on any computation path of the program. For example,

$$\texttt{r1 = y; r2 = (r1 == 0) ? x : x} \longrightarrow \texttt{r1 = y; r2 = x}$$

Reordering normal memory accesses reorders independent non-volatile memory accesses:

$$\texttt{r1 = y; x = 1} \longrightarrow \texttt{x = 1; r1 = y}$$

Redundant read after read elimination reuses a value from a previous non-volatile read of the same variable:

$$\texttt{r1 = x; r2 = x} \longrightarrow \texttt{r1 = x; r2 = r1}$$

Redundant read after write elimination replaces a non-volatile read with a local value if the value is known from a previous write:

$$\texttt{x = 1; r2 = x} \longrightarrow \texttt{x = 1; r2 = 1}$$

Irrelevant read elimination removes a non-volatile read if its value is not used:

$$\texttt{r1 = x; r1 = 1} \longrightarrow \texttt{r1 = 1}$$

Irrelevant read introduction is an inverse of the previous transformation:

$$\texttt{x = 1; r2 = x} \longrightarrow \texttt{x = 1; r2 = 1}$$

Redundant write before write elimination removes an overwritten non-volatile write:

$$\texttt{x = 1; x = 2} \longrightarrow \texttt{x = 2}$$

Redundant write after read elimination removes a write if it is known that the location has the value that is being written:

$$\texttt{r1 = x; x = r1} \longrightarrow \texttt{r1 = x}$$

Roach-motel reordering reorders a non-volatile memory access with a following acquire action (i.e., a lock or a volatile read) or with a previous release action (i.e., an unlock or a volatile write):

$$\texttt{x = 1; lock m} \longrightarrow \texttt{lock m; x = 1}$$

External action reordering swaps an I/O action with an independent memory access:

$$\texttt{x = 1; print 0} \longrightarrow \texttt{print 0; x = 1}$$

```
      x = y = 0                        x = y = 0                 x = y = 0
 -----------------------          -------------------        -----------------
 r1=x | r2=y                      r1=x | r2=y                r1=x | x=1
 y=r1 | x=(r2==1)?y:1     ⟶       y=r1 | x=1          ⟶      y=r1 | r2=y
      | print r2                       | print r2                 | print r2
```

Figure 2.1: Hotspot JVM's transformations violating the JMM.

2.1.2 Proving Validity of Transformations

To establish the validity results, we have developed a trace semantic framework where the semantics of a program is a set of traces of its threads. For example, the meaning of the program

```
Thread 1: x := 1; y := 1; v := 1;
Thread 2: if (v > 0) print x + y;
```

is essentially the set of traces

$$\{[\mathrm{S}(\texttt{T1}), \mathrm{W}(\texttt{x}, 1), \mathrm{W}(\texttt{y}, 1), \mathrm{W}(\texttt{v}, 1)]\} \cup \{[\mathrm{S}(\texttt{T2}), \mathrm{R}(\texttt{v}, v)] \mid v \leq 0\} \cup$$
$$\{[\mathrm{S}(\texttt{T2}), \mathrm{R}(\texttt{v}, t), \mathrm{R}(\texttt{x}, u), \mathrm{R}(\texttt{y}, v), \mathrm{X}(u + v)] \mid t > 0, u, v \in \mathbb{Z}\}$$

where $\mathrm{S}(t)$ is a start action of thread t, $\mathrm{R}(x, v)$ (resp. $\mathrm{W}(x, v)$) is a read of value v from (resp. a write of value v to) shared location x, and $\mathrm{X}(v)$ is an output of v.

We describe the program transformations as relations on sets of traces. To show validity in the Java Memory Model, we take an execution of the transformed program and construct an execution of the original program with the same behaviour. We obtain the execution by 'undoing' the transformations on the traces of threads in the execution and construct an execution of the original program together with the committing sequence for the execution.

To prove the DRF guarantee for the transformations, we use a similar overall procedure, but we can assume that the original program was data race free and we only need to find an interleaving of the original program with the same behaviour. We also show that all the considered transformations (except the irrelevant read introduction) preserve data race freedom. This implies that any composition of the transformations is safe. For details, see [15].

2.1.3 Invalid Transformations

Unfortunately, we have discovered serious problems in the JMM—several important transformations are invalid in the Java Memory Model. The transformation in Figure 2.1 illustrates this on an example. The first program in the figure cannot print 1 in the Java Memory Model (for details, see [16]). A typical optimising compiler may reuse the value of `y` in `r2` and transform

$$\texttt{x=(r2==1)?y:1} \;\rightarrow\; \texttt{x=(r2==1)?r2:1} \;\rightarrow\; \texttt{x=1}$$

as shown by the first transformation in Figure 2.1. Then it may reorder the write to `x` with read of `y`, yielding the last program in Figure 2.1. Observe that this transformed program can print 1 using the interleaving `x=1`, `r1=x`, `y=r1`, `r2=y`, `print r2`.

This is serious flaw in the JMM because the model was designed to validate such transformations. We have contacted the authors of the model and they acknowledged the problem and confirmed that it would not be easy to fix.

It is less clear whether this is a serious problem in practice. On one hand, even Sun's HotSpot JVM performs the optimisation from Figure 2.1 and programs can exhibit behaviours that are forbidden by the specification. For details, see [17]. On the other hand, we believe that the JVM does not perform any optimisation that violates the DRF guarantee or out-of-thin-air guarantees (note that there are data races in the programs in Figure 2.1). This claim is strongly supported by our investigation of transformations in the DRF guarantee described in the previous subsection. Therefore, programmers probably need not worry about the semantics of their data race free programs—Sun's JVM still seems to provide the strong guarantees of the JMM even though it does not implement the memory model precisely.

2.2 BicolanoMT

BicolanoMT is the extension of Bicolano – the formalization of the Java Virtual Machine Specification developed by D. Pichardie et al. [46, 40] – with multithreading. In particular, we add instructions `monitorenter` and `monitorexit`, and we provide a semantics for the (non-deprecated) native methods related to concurrency (`start`, `join`, `wait`, `notify` and `notifyAll`). Moreover, we model the interrupt mechanism of Java.

The main benefits provided by the BicolanoMT formalization include the following:

- The formalization is faithful to Java, i.e., a bytecode class can be directly mapped into the Coq representation that we use, and no artificial code transformations – such as wrapping the body of a synchronized method in a synchronized statement block – are necessary.
- Since we describe the Java semantics at bytecode level, our formalization has the right level of granularity to handle multithreading, i.e., interleaving of sequential instructions is described naturally. Moreover, in contrast to source code level formalizations (like [38]), we do not have to add special tags to mark that execution is within a synchronized block.
- Sequential Bicolano has been used to show soundness of program logics and type systems for secure information flow. Our extension for multithreading allows extending these soundness results for multithreaded programs (e.g., an extension of the program logic with rely-guarantee, or a formalization of a type system for secure information flow of multithreaded programs [7], where the sequential part is already formalized [6]). Moreover, using Coq's extraction mechanism, we can get the implementation of a verified verification condition generator or type checker for free.

Based on the *Data Race Freeness* guarantee provided by the Java memory model (JMM) [39], we can consider an interleaving semantics only, abstracting away from all the details of the JMM. This guarantee ensures that all *correctly synchronized* programs only exhibit behaviors described by an interleaving semantics. Program analyses and logics proved with this semantics will thus be valid for correctly synchronized programs. We resort to existing data race detection static analyses to reject incorrectly synchronized programs [18, 41, 42] (see also the revival of the RCC checker described in the intermediate deliverable). Interestingly, programs containing data races can also be treated with our formalization, provided that the semantics of these races is described by some interleaving of the threads (i.e., *benign data races*).

On a more technical side: to separate the multithreading concerns from those of the sequential Bicolano formalization, and to better adapt to future changes in Bicolano — upon which BicolanoMT depends — we have applied the extension framework proposed by Czarnik and Schubert [14]. Their framework enables the extension of the Bicolano semantics with additional information and/or additional behavior. In particular, we use the vertical extension mechanism described there, adding extra information to the state definition, and adding new cases to the step relation. Note that the extensions are constructive, in the sense that they only add consistent behaviors.

In what follows we will give a flavor of the BicolanoMT formalization, and highlight its most important constituents. This is by no means a complete report on BicolanoMT. We refer the interested reader to [31] for a more detailed description.

2.2.1 Data Structures

Several data structures present in Bicolano must be extended in order to account for multithreading. Here we present the most important changes.

```
Module Type HEAP_MT.
 Declare Module Heap : HEAP.
 Import Heap.
 Parameter lock : t -> Location -> ThreadId.t -> t.
 Parameter unlock : t -> Location -> ThreadId.t -> t.
 Parameter getLockLevel : t -> Location -> ThreadId.t -> nat.
 Parameter getWaitLockLevel : t -> ThreadId.t -> nat.
 Parameter getWaitset : t -> Location -> ThreadSet.t.
 Inductive waitFor_and_unlock h loc tid h' : Prop := ...
End HEAP_MT.
```

Figure 2.2: Fragment of Multithreaded Heap definition

Firstly, we must adapt the sequential definition of the program to several concurrent threads. Thus, we define the `Thread` abstract module type that contains the type definition of threads. Threads can be either in normal state or in exceptional state; their difference being that exceptional threads *current frame* is exceptional (i.e., it contains a raised exception). The callstack of each thread is recorded in this data structure, as well as its execution state (one of: `runnable`, `blocked`, `waiting` or `terminated`), and a flag recording whether the thread has been interrupted (via the `Thread.interrupt()` method for example).

```
Module Type THREAD.
 Inductive t : Type :=
   normal : Frame.t -> CallStack.t -> ThreadState.t -> Interruption -> t
 | exception : ExceptionFrame.t -> CallStack.t -> ThreadState.t ->
               Interruption -> t.
End THREAD.
```

To keep track of the several threads, we specify the `ThreadMap` data structure. The thread map basically is a mapping from thread identifiers to thread instances.

The most important modification to the sequential Bicolano framework is the `Heap` data structure, which is replaced by the `HeapMT` structure, since the multithreaded version of the heap must account for monitor instructions, namely `monitorenter` and `monitorexit`, as well as instructions regarding the `wait()` and `notify()` native methods provided by the `Thread` class. However, note that this definition of `HeapMT` reuses the definition of `Heap`. We have specified all these operations by means of an abstract type that defines the more abstract `lock` and `unlock` operations that will be the building blocks of the semantics of the Java multithreaded instructions. The most relevant part of the `HeapMT` specification can be found in Figure 2.2.

Finally, the last definition that needs to be augmented is the `State`. The multithreaded state (`MT_State`) keeps record of a single *current* thread that is the one being scheduled for execution, and a thread map. It also provides means to update (or re-schedule) a new thread. An important remark is that the `MT_State` data structure is conservative, in the sense that it reconstructs the multithreaded state from a sequential Bicolano state, and the information provided by the thread map. This is required by the extensional framework of Czarnik and Schubert.

2.2.2 Interleaving Semantics

The multithreaded interleaving semantics builds on top of the sequential semantics. It performs an extra check; namely that the thread is in `runnable` state and that is has not been `interrupted`. Also, semantic cases for all the multithreaded instructions are given (which include exceptional cases). The list of instructions added to the sequential semantics contains: `native_start`, `monitorenter`, `monitorexit`, `invoke_synch` corresponding to the invocation of a synchronized method, `native_wait`, `native_notify`, `native_notifyAll`, and `native_interrupt`. The `native_` prefix is given to instructions that are not part

```
| monitorenter_nonblocking_mtstep_ok : ...
   instructionAt m pc = Some monitorenter -> ...
   lockable h loc tid ->
   h' = Heap_mt.lock h loc tid -> ...
   Support.additional_step p sst (St h' f' cs) sst' ->
   tmap' = ThreadMap.update tmap tid (Tr f' cs runnable it) ->
   Some mtst' = update mtst sst' tid tmap' ->
   mt_step p mtst mtst'
```

Figure 2.3: Step `monitorenter`

of the instruction set, but rather implemented as native methods. Since these methods are crucial for giving a sensible semantics to multithreaded Java we have included them by means of these extra instructions.

For brevity we will only explain the `monitorenter` and `monitorexit` specification, as a paradigmatic example. The specification of the other relevant instructions can be found in [31].

The `monitorenter` and `monitorexit` bytecode instructions lock and unlock Java monitors. Both take as parameter a reference whose lock must be acquired or released, respectively. Figure 2.3 shows the formalization of a `monitorenter` instruction that succeeds to acquire the lock[1]. First, the heap is checked for the state of the lock. If it is free or acquired by the same thread, `lockable` holds. In that case the lock is acquired by the thread, and the heap and state are updated. The case in which the lock is not free is not shown, but the main difference is that the heap is not updated, and the thread state of the thread is changed to `blocked`, meaning that the thread is only able to re-attempt to acquire the lock. BicolanoMT also specifies all the exceptional cases documented in the JVM Specification. The formalization of `monitorexit` is similar, but the predicate `locked_by` is checked instead, and an `IllegalMonitorStateException` is thrown when the lock is not held.

Finally, it is important to notice that all the exceptional cases, as well as all possible interactions of the different thread states and instructions (as well as native methods) are specified in the semantics. Therefore the number of cases considered is significant (around 500 lines of Coq specification). However, we do not formalize thread groups, timing and class loading, as these are not allowed in the MIDP framework, which is the target domain for verification techniques developed within MOBIUS.

[1]Only the relevant conditions are shown.

Chapter 3

Separation Logic for a Multithreaded Java-like Language

This chapter describes the development of a sound and practical verification method to reason about multithreaded applications in a Java-like language.

Recently, separation logic [48] made a dent into program verification by taming problems related with aliasing. Initially, separation logic has been applied to simple while languages [48, 8]. Later, separation logic has been lifted to more realistic programming languages, including parallel languages [44, 20] and object-oriented languages [45].

The preliminary results described in the intermediate deliverable evolved into an extension of permission-based separation logic [25, 26, 24, 27, 22, 23] to reason about applications written in a Java-like language. Our program logic features an expressive and flexible specification language, that subsumes the need for explicit keywords, as presented in the intermediate report. Previous work on object-oriented separation logic existed [45], however, it was restricted to sequential programs. We extended this work in several ways. First, we lifted it to deal with Java's fork/join style of parallelism [25, 26]. Second, we crafted new proof rules for reentrant locks, i.e., locks that can be acquired more than once [22, 23]. Reentrant locks are a common primitive of modern programming languages such as Java. We applied our system against various flavors of a concurrent iterators [24, 27]. Notice that a similar technique has recently been added to the Boogie methodology [37]. However, instead of using separation logic directly, they encode fractional permissions into BoogiePL.

The next chapter shows how the program logic can be used in derived ways: *(1)* to specify usage protocols of multithreaded classes and to check such protocols against method contracts, and *(2)* to apply provably correct program optimizations.

3.1 Separation Logic for a Java-like Language with Fork/Join

In [25, 26], we present new proof rules for a Java-like language with threads and fork/join as concurrency primitives. In order to facilitate concurrent reads we employ fractional permissions [11]. Our rules allow multiple threads to join on the same thread, in order to read-share the dead thread's resources. This is not possible with a lexically scoped parallel composition operator. Below we present our rules informally. To help the reader understand these rules, we recall that, in separation logic, formulas also denote permissions to access part of the heap. In particular, a thread's precondition (specified for the special method `run`) represents the part of the heap that can legally be accessed by a thread during its life time, while a thread's postcondition represents the part of the heap that can be transferred to other threads after a thread has terminated. Our rules for fork and join are as follows (where t denotes a thread):

$$\frac{F \text{ is } t\text{'s precondition}}{\{F\}t.\texttt{fork()}\{\texttt{true}\}}\,(\text{Fork}) \qquad \frac{F \text{ is } t\text{'s postcondition}}{\{\texttt{Join}(t,\pi)\}t.\texttt{join()}\{\pi \cdot F\}}\,(\text{Join})$$

The rule for `fork` expresses that, when a thread is created, it obtains permissions to write to the heap from its parent thread. The rule for `join` expresses that, when a dead thread is joined, it hands back permissions to write to the heap to the joining thread. This is expressed by the joining permission $\texttt{Join}(t, \pi)$ and the scaling operation $\pi \cdot F$. In these formulas, π is a fraction in $(0, 1]$. $\texttt{Join}(t, \pi)$ expresses that the joining thread will obtain fraction π of t's postcondition. For example, if π is 1, the joining thread obtains the entire part of the heap that was accessible by the dead thread. This is expressed by the scaling operation $\pi \cdot F$ which represents fraction π of formula F. If $\pi = 1$, then $\pi \cdot F = F$.

To support data abstraction and recursive data types, we use abstract predicates [45]. Abstract predicates are declared at the class level, à la JML [36], with the keywords `pred` and `group`. Abstract predicates declared with the `group` keyword must satisfy a split/merge axiom. The split/merge axiom is used to define the scaling operation $\pi \cdot F$ used in (Join)'s postcondition. Class `Point` exemplifies an abstract predicate satisfying the split/merge axiom:

```
class Point extends Object{
 int x,y;
 //@ pred state<perm p> = Perm(x,p) * Perm(y,p);
 //@ requires state<1>; ensures state<1>;
 void shiftBy(int x,int y){
  this.x += x; this.y += y;
 }
}
```

Class `Point` declares an abstract predicate `state`. The *resource conjunction* $F * G$ expresses that F and G describes *disjoint* parts of the heap. The `Perm` predicate `Perm(x,p)` asserts permission to read (if `p< 1`) or write (if `p = 1`) to `this.x`. The `state` predicate represents permission `p` to access fields `x` and `y` of instances of class `Point`. The `state` predicate is parameterized by permission `p` to distinguish between read and write operations: `state<1>` represents permission to read *and write* to fields `x` and `y`, while `state<p>` for any `p < 1` represents permission to *readonly* access fields `x` and `y`. À la JML, the keyword `requires` indicates the beginning of the *precondition* (aka *requires-clause*), and the keyword `ens` the beginning of the *postcondition* (aka *ensures-clause*). Because method `shiftBy` requires write access to fields `x` and `y`, it has `state<1>` as its precondition. In [25], we show how abstract predicates satisfying the split/merge axiom are useful to specify that objects can be shared among multiple threads.

3.2 Separation Logic for Java's Reentrant Locks

In [22, 23], we present new proof rules for a Java-like languages with reentrant locks. Reentrant locks can be acquired twice or more and, as already mentioned, are a common primitive of modern programming languages such as Java. Reentrant locks are considered to be a facility for programmers because programmers do not have to worry if a lock is acquired or not before acquiring it. While reentrant locks facilitates the task of programming, they complicate the underlying proof systems: proof systems for reentrant locks have to distinguish between initial and reentrant acquiring of locks.

In separation logic [44], the programmer formally associates locks with pieces of heap space, and the verification system ensures that a piece of heap space is only accessed when the associated lock is held. The piece of heap space guarded by a lock is called the lock's *resource invariant.* Our first achievement was to modularly specify the resource invariant of objects. In Java, each object has an associated reentrant lock, its *object lock.* Naturally, the resource invariant that is associated with an object lock is specified in the object's class. For subclassing, we need to provide a mechanism for extending resource invariants in subclasses in order to account for extended object state. To this end, we represent resource invariants as abstract predicates [45]: resource invariants are distinguished abstract predicates named `inv`. They have a default definition in the `Object` class and are meant to be extended in subclasses:

```
class Object { ...  pred inv = true; ...  }
```

The resource invariant o.`inv` can be assumed when o's lock is acquired non-reentrantly and must be established when o's lock is released with its reentrancy level dropping to 0. Regarding the interaction with subclassing, there is nothing special about `inv`. It is treated just like other abstract predicates.

In separation logic for single-entrant locks [44], locks can be acquired without precondition. For reentrant locks, on the other hand, it seems unavoidable that the proof rule for acquiring a lock distinguishes between initial acquires and re-acquires. This is needed because it is quite obviously unsound to simply assume the resource invariant after a re-acquire. Thus, a proof system for reentrant locks must keep track of the locks that the current thread holds. To this end, we introduce two new formulas: `Lockset`(S) and S `contains` o (where S denotes a *multiset* of objects). Informally, `Lockset`(S) means that S is the multiset of locks held by the current thread. Multiplicities record the current reentrancy level i.e., the number of times locks have been acquired. S `contains` o means that multiset S contains object o.

While `lockset` and `contains` track the state of threads, we also need formulas to track the state of locks. Like class invariants must be initialized before method calls, resource invariants must be initialized before the associated locks can be acquired. In O'Hearn's simple concurrent language [44], the set of locks is static and initialization of resource invariants is achieved in a global initialization phase. This is not possible when locks are created dynamically. Conceivably, we could tie the initialization of resource invariants to the end of object constructors. However, this is problematic because Java's object constructors are free to leak references to partially constructed objects (e.g., by passing `this` to other methods). Thus, in practice we have to distinguish between initialized and uninitialized objects semantically. Furthermore, a semantic distinction enables late initialization of resource invariants, which can be useful for objects that remain thread-local for some time before getting shared among threads. To support flexible initialization of resource invariants, we introduce two more formulas: o.`fresh` and o.`initialized`. o.`fresh` means that o's resource invariant is not yet initialized. o.`initialized` means that o's resource invariant has been initialized.

3.2.1 Initializing resource invariants.

The `fresh`-predicate is introduced as a postcondition of `new`:

$$\frac{}{\{\texttt{true}\}o = \texttt{new()}\{o.\texttt{fresh} * \ldots\}}\ (\text{New})$$

The specification command o.`commit` triggers o's transition from the `fresh` to the `initialized` state, provided o's resource invariant is established:

$$\frac{}{\begin{array}{c}\{\texttt{Lockset}(S) * o.\texttt{inv} * o.\texttt{fresh}\}\\ o.\texttt{commit}\\ \{\texttt{Lockset}(S) * \texttt{!}(S\ \texttt{contains}\ o) * o.\texttt{initialized}\}\end{array}}\ (\text{Commit})$$

3.2.2 Locking and unlocking.

There are two rules each for locking and unlocking, depending on whether or not the `lock`/`unlock` is associated with an initial entry or a reentry (where $\cdot$ denotes multiset union):

$$\frac{}{\begin{array}{c}\{\texttt{Lockset}(S) * \texttt{!}(S\ \texttt{contains}\ o) * o.\texttt{initialized}\}\\ o.\texttt{lock()}\\ \{\texttt{Lockset}(S \cdot o) * o.\texttt{inv}\}\end{array}}\ (\text{Lock}) \qquad \frac{}{\begin{array}{c}\{\texttt{Lockset}(S)\}\\ o.\texttt{lock()}\\ \{\texttt{Lockset}(S \cdot o)\}\end{array}}\ (\text{Re-Lock})$$

The rule (Lock) applies when lock o is acquired non-reentrantly, as expressed by the Hoare triple's precondition: `Lockset`(S) `* !`$(S$ `contains` $o)$. The precondition o.`initialized` makes sure that *(1)* threads only acquire locks whose resource invariant is initialized, and *(2)* no null-error can happen (because initialized values are non-null). The postcondition adds o to the current thread's lockset, and assumes o's resource invariant. The rule (Re-Lock) applies when a lock is acquired reentrantly.

Next we present the rules for releasing locks:

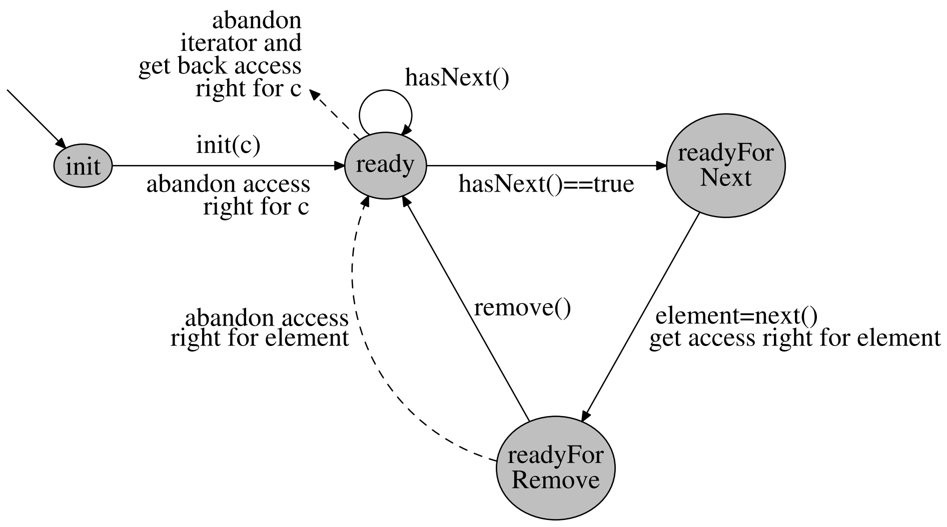

Figure 3.1: Usage Protocol of the `Iterator` interface

$$\frac{}{\{\texttt{Lockset}(S \cdot o \cdot o)\}\ o.\texttt{unlock()}\ \{\texttt{Lockset}(S \cdot o)\}}\ \text{(Unlock)} \qquad \frac{}{\{\texttt{Lockset}(S \cdot o) * o.\texttt{inv}\}\ o.\texttt{unlock()}\ \{\texttt{Lockset}(S)\}}\ \text{(Re-Unlock)}$$

The rule (Unlock) applies when o's current reentrancy level is at least 2, and (Re-Unlock) applies when o's current reentrant level might be 1.

3.3 Resource Usage Protocols for Iterators

In [24, 27], we study various flavors of a concurrent iterator and its usage protocol and show how this protocol can be expressed in our extension of separation logic [25]. Usage protocols for iterators are meant to prevent so-called *concurrent modifications* of the underlying collection: while an iterator over a collection is in progress, the collection should not get modified by other actions that interleave with the iteration[1]. Concurrent modifications need to be prevented, because they may temporarily break invariants of the collection, resulting in iterators seeing collections in inconsistent states. Concurrent modifications can be caused both by modifications of the collection itself (i.e., adding or removing collection elements), or in some cases by modification of the collection elements (i.e., resetting fields of collection elements).

In Java, the `Collection` and `Iterator` interfaces are as follows:

```
interface Collection{             interface Iterator{
  Iterator iterator();             boolean hasNext();
}                                  Object next();
                                   void remove();
                                  }
```

Figure 3.1 depicts Interface `Iterator`'s protocol: `hasNext()` must be called first, then `next()` should be called if `hasNext()` returned true before, and then `remove()` may be called if the iterator is read-write (it can read and write to the iteratee), and so on. Here is the `Iterator` interface that formalizes Figure 3.1:

```
interface Collection {
    //@ requires this.space; ensures result.ready;
    Iterator/*@<this>@*/ iterator();
```

[1]Such interleaving actions may execute in the same thread as the iterator.

```
}

interface Iterator/*@<Collection iteratee>@*/ {
    //@ pred ready;
    //@ pred readyForNext;
    //@ pred readyForRemove<Object element>;
    //@ axiom ready -* iteratee.space;
    //@ axiom (fa Object e)(readyForRemove<e> * e.space -* ready);
    //@ requires ready; ensures ready & (result -* readyForNext);
    boolean hasNext();
    //@ requires readyForNext;
    //@ ensures readyForRemove<result> * result.space;
    Object next();
    //@ requires readyForRemove<Object>; ensures ready;
    void remove();
}
```

This interface declares three abstract predicates `ready`, `readyForNext` and `readyForRemove` (where the keyword `fa` denotes universal quantification). Interface implementations must define these predicates in terms of concrete separation logic formulas. The predicate definitions must be such that the two *class axioms* are tautologically true, and that the methods satisfy their contracts (after replacing the abstract predicate symbols by their concrete definitions). Each class extends a generic predicate `space`, which has a default definition in the `Object` class. This predicate should define the heap space that is associated with an object — often consisting of the object fields only, but in the case of collections sometimes also including the object spaces of the collection elements. Reference types and predicates may be parameterized by values. For instance, the `Iterator` class is parameterized by the collection, and the `readyForRemove` predicate is parameterized by the collection element that is ready to be removed. The `&`-operator represents *choice.* If F & G holds, then F and G are available, but are interdependent: using either one of them destroys the other, too. The `&`-operator can be used to represent non-deterministic state transitions, as exhibited in the postcondition of `hasNext()`. The *resource implication* F -* G grants the right to consume F yielding G.

The protocol above is the basis of the protocols for various flavors of iterators [24]. We have shown how to support *(1)* multiple read-only iterators over the same collection, *(2)* unrestricted access to immutable collection elements, and *(3)* collections where the element access rights stay with the elements rather than being governed by the collection. This confirmed our belief that our variant of separation logic is expressive and powerful enough to handle real world case studies.

Chapter 4

Other Uses of the Program Logic

4.1 Specifying and Checking Protocols of Multithreaded Classes

4.1.1 Specifying protocols

In [33], we extend earlier work on specifying and checking protocols of classes [12] to deal with a variant of generic classes (i.e., classes parameterized by Java values or specification values) and multithreaded classes (i.e., classes such that multiple threads may execute a method simultaneously on instances of this class). To show that method contracts are correct w.r.t. protocols, we generate programs that must be proven correct (in analogy with traditional program verifiers [13] that generate proof obligations). Because little support currently exists to help writing method contracts, our technique helps programmers to check their contracts early in the development process.

Protocols are specified by means of regular expressions that indicate which methods can be called on an object and in which order. As a general observation (cf. [12]), we note that many classes must be used in a specific way by clients: for these classes it is important to be able to concisely express allowed method call sequences. For example, clients of Java's `StreamBuffer` interface must call method `read()` zero or more time and then call method `close()` exactly *once*. This can be concisely specified by the following protocol:

```
read()*, close()
```
(4.1)

We propose to specify such protocols at the class level with the keyword `protocol`. An advanced example is the protocol for Java's `Iterator` outlined in Section 3.3: `hasNext()` must be called first, then `next()` should be called if `hasNext()` returned true before, and then `remove()` may be called if the iterator is read-write (it can read and write to the iteratee), and so on. To express that iterators may have write-access to the iteratee, we parameterize interface `Iterator` by a permission `p` (cf. [25]). If `p` is instantiated by `1`, one obtains a read-write iterator, otherwise a read-only iterator. Now, one can precisely and concisely express the `Iterator`'s protocol as follows:

```
interface Iterator<perm p> {
 protocol (v=hasNext(), v?(next(), p==1?(remove())))*;
 boolean hasNext();
 Object next();
 void remove();
}
```

In the protocol above, `v?(next(), p==1?(remove()))` expresses that `next` and `remove` can be called only if the previous call to `hasNext` returned true (in analogy with Java's syntax for inlined conditional i.e., "b ? . . . : . . ."). Similarly, `p==1?(remove())` expresses that `remove` may be called only if the iterator is read-write (i.e.,`p` is `1`). We believe that such a formal specification (compared to Java's informal documentation) would help clients to use `Iterator`s in a disciplined way.

4.1.2 Checking protocols

Finally, we propose a technique for checking that protocols are correct w.r.t. method contracts. The key observation is that, because protocols specify in which order method calls *must* be done, it should be checked that method contracts *adhere* to such an order. To check protocols, we generate programs that can be verified with standard techniques using our variation of separation logic [25, 26, 22, 23]. If the generated program cannot be proven correct, it means that the method contracts are incorrect w.r.t. the protocol: some (client-provided) programs will fail to verify. As an example, to check that method contracts of interface `Iterator` adhere to the `Iterator`'s protocol, one must verify the following program (where `*` models non-deterministic choice):

```
requires ... * Perm(i.v,1);
ensures  true;
void checkAdherence(Iterator<p> i){
 ...  // initialization (not shown in Iterator's interface)
 while(*){
  i.v = hasNext();
  if(*){
   assume(i.v); Object o = i.next();
    if(*){ assume(i.p==1); i.remove(); }
  }
 }
}
```

One caveat is that the precondition of method contracts considered must be *receiver splittable* i.e., they must have the form F *op* G (where *op* = {`*`, `&`}) and no method parameter (except `this` and `result`) occurs in F. The reason for this restriction is that, when generating programs, we cannot (easily) provide method parameters fulfilling the part of the method preconditions which is relevant to method parameters. The precondition allows to syntactically split method contracts into a part that concerns the receiver `this` (F) and a part that concerns the parameters (G). Then, when checking adherence of protocols, the part of the preconditions relevant to parameters (G) is dropped. We believe that, in practice, most method preconditions are receiver splittable. Our belief is supported by the fact that all examples that are used to illustrate object-oriented separation logic [45, 25, 26, 22, 23] are receiver splittable.

4.2 Automatic Parallelization and Optimization of Programs by Proof Rewriting

Since writing parallel software is notoriously harder than writing sequential software, inferring parallelism automatically is a possible solution to the challenges faced by software developers. A well-known technique for inferring parallelism is to detect pieces of programs that access disjoint parts of the heap. In earlier work, various pointer analyses have been used to achieve this for programs manipulating simple data structures and arrays, see e.g., [28, 19, 21].

This section describes a new technique to infer parallelism from proven programs. Instead of designing ad-hoc analysis techniques, we use separation logic [48] to analyze programs before parallelizing them. We use separation logic's `*` operator – which expresses disjointness of parts of the heap – to detect potential parallelism. Compared to [28, 19, 21], using the `*` operator avoids relying on reachability properties. This permits to discover disjointness of arbitrary data structures, paving the way to parallelize and optimize object-oriented programs proven correct with separation logic [45]. For the moment, this work is not yet adapted to our variant of separation logic, instead it uses a classical variant of separation logic and the parallel operator. However, it is possible to apply the ideas presented below also to our specific flavour of separation logic.

Contrary to most earlier work (that manipulate formulas representing proof obligations [5, 43] or manipulate labelled heaps [47]), our algorithms manipulate proof trees representing derivations of Hoare triplets. The overall procedure is as follows: we generate a proof tree $\mathcal{P}$ of a program C, then we rewrite $\mathcal{P}, C$ into $\mathcal{P}', C'$ such that $\mathcal{P}'$ is a proof of C' and C' is a parallelized and optimized version of C. Proof trees are generated using a modified version of Smallfoot [9], and the proof tree rewrite system is implemented in Tom [4].

Our algorithm for rewriting proof trees focuses on two rules of separation logic: the (Frame) and the (Parallel) rules. First, the (Frame) rule [48] allows reasoning about a program in isolation from its environment, by focusing only on the part of the heap that the program accesses. Second, the (Parallel) rule [44] allows reasoning about parallel programs that access disjoint parts of the heap.

$$\frac{\{F\}C\{F'\}}{\{F * G\}C\{F' * G\}}\ \text{(Frame)} \qquad \frac{\{F\}C\{G\} \qquad \{F'\}C'\{G'\}}{\{F * F'\}C \| C'\{G * G'\}}\ \text{(Parallel)}$$

The basic idea of our reasoning is depicted by the following rewrite rule:

$$\frac{\dfrac{\{F\}C\{G\}}{\{F * F'\}C\{G * F'\}}\ \text{(Frame)} \qquad \dfrac{\{F'\}C'\{G'\}}{\{G * F'\}C'\{G * G'\}}\ \text{(Frame)}}{\{F * F'\}C;\ C'\{G * G'\}}\ \text{(Seq)}$$

$$\downarrow \underline{\text{Parallelize}}$$

$$\frac{\{F\}C\{G\} \qquad \{F'\}C'\{G'\}}{\{F * F'\}C \| C'\{G * G'\}}\ \text{(Parallel)}$$

This diagram should be read as follows: Given a proof of the sequential program $C;\ C'$ we rewrite this proof into a proof of the parallel program $C \| C'$. If the initial proof tree is valid, this rewriting yields a valid proof tree because the leaves of the rewritten proof tree are included in the leaves of the initial proof tree.

Our procedure differs from recent and concurrent work [47] on three main points: *(1)* instead of attaching labels to heaps, we use the (Frame) rule to statically detect independent parts of the program, leading to a technically simpler procedure; *(2)* we express optimizations by rewrite rules on proof trees, allowing us to feature other optimizations than parallelization and to use different optimization strategies; and *(3)* we have an implementation.

To develop our approach, we carefully redesigned the standard proof rules from Berdine et al. [9] to adapt them to our rewrite rules, in particular so that they compute frames with enough precision to be able to apply the rewrite rules. Based on the adapted proof rules, we derive sound rewrite rules from proof trees to proof trees that yield optimized programs. The (Frame) rule is the central ingredient of this procedure.

However, once (Frame)s have been computed with enough precision, the proof trees obtained are often still inadequate for our procedure to work. This is due to the particular shape of the proof trees obtained. Generally speaking, given a program of the form $C_0;\ C_1;\ C_2$, proof trees obtained from existing program verifiers such as Smallfoot have the following shape:

$$\frac{\dfrac{\dots}{\{\dots\}C_0\{\dots\}}\ \text{(Frame } F\text{)} \qquad \dfrac{\dfrac{\dots}{\{\dots\}C_1\{\dots\}}\ \text{(Frame } G\text{)} \qquad \dots}{\{\dots\}C_1;\ C_2\{\dots\}}\ \text{(Seq)}}{\{\dots\}C_0;\ C_1;\ C_2\{\dots\}}\ \text{(Seq)}$$

The problem with the tree shown above is that the successive frames F and G are often redundant. For example, if the commands $C_0;\ C_1$ and C_2 access different part of the state, F and G will represent a similar part of the state. For our procedure to work, we need to remove redundancies in frames, by factorizing them. Figure 4.1 shows a rewrite rule for *factorizing frames*. This rewrite rule fires if two successive frames F_f and G_f have a common part F_c. The proof tree obtained after factorization frames the common part of the frame below $C;\ C'$. Once frames have been factorized, the $\underline{\text{Parallelize}}$ rule shown above can often infer parallelism [32].

Considered optimizations are parallelization, early disposal, late allocation, early lock releasing, and late lock acquiring.

$$\dfrac{\dfrac{\{F_a\}C\{F_p\}}{\{F_a * F_f\}C\{F_p * F_f\}}\,(\text{Frame } F_f) \qquad \dfrac{\dfrac{\{G_a\}C'\{G_p\}}{\{G_a * G_f\}C'\{G_p * G_f\}}\,(\text{Frame } G_f) \qquad \{G_p * G_f\}C''\{F'\}}{\{G_a * G_f\}C';\ C''\{F'\}}\,(\text{Seq})}{\{F_a * F_f\}C;\ C';\ C''\{F'\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{FactorizeFrames}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\{F_a\}C\{F_p\}}{\{F_a * F_{f_0}\}C\{F_p * F_{f_0}\}}\,(\text{Frame } F_{f_0}) \qquad \dfrac{\{G_a\}C'\{G_p\}}{\{G_a * G_{f_0}\}C'\{G_p * G_{f_0}\}}\,(\text{Frame } G_{f_0})}{\{F_a * F_{f_0}\}C;\ C'\{G_p * G_{f_0}\}}\,(\text{Seq})}{\{F_a * F_f\}C;\ C'\{G_p * G_f\}}\,(\text{Frame } F_c) \qquad \{G_p * G_f\}C''\{F'\}}{\{F_a * F_f\}C;\ C';\ C''\{F'\}}\,(\text{Seq})$$

Guard: $F_f \Leftrightarrow F_{f_0} * F_c$ and $G_f \Leftrightarrow G_{f_0} * F_c$

Figure 4.1: Rewrite rule to factorize applications of (Frame)

Chapter 5

Conclusions

This document describes the final results of Task 3.3 of the MOBIUS project on verification of multithreaded applications. It describes contributions on two different topics. First, we have contributed to the formal description of the semantics of multithreaded Java applications, by studying the semantics of the Java Memory Model, and by giving a detailed and complete formalization of relevant bytecode instructions and native methods, and defining an appropriate interleaving semantics. Second, we have developed a program logic, based on permission-based separation logic, for multithreaded applications, that considers object-orientation, the fork-join mechanism and reentrant locks, i.e., Java's most relevant features. The program logic allows to write and verify elegant and expressive specifications. Moreover, it can also be used for other applications, such as checking whether a program adheres to a usage protocol, and to apply provably correct program optimizations (including parallelization).

Further investigation is still needed to see whether the program logic can be used for automatic verification. Currently, we are working on a simple refutation procedure, that quickly can decide whether a program can *not* be proven correct. This will be reported in the forthcoming PhD thesis of Clément Hurlin, along with more additional material on the topics of chapters 3 and 4.

Within Task 5.3, we plan to use permission-based separation logic to specify the typical concurrency patterns that occur in midlets. We have shown that many general concurrency patterns can be specified and verified with our approach, but we did not consider the specific midlet usage of concurrency yet.

Bibliography

[1] E. Ábrahám. *An Assertional Proof System for Multithreaded Java - Theory and Tool Support.* PhD thesis, University of Leiden, 2004.

[2] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. Tool-supported proof system for multithreaded Java. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, number 2852 in Lecture Notes in Computer Science, pages 1–32. Springer-Verlag, 2003.

[3] D. Aspinall and J. Ševčík. Formalising Java's data-race-free guarantee. In *Theorem Proving in Higher Order Logics: Proceedings of the 20th International Conference TPHOLs 2007*, Lecture Notes in Computer Science 4732, pages 22–37. Springer-Verlag, 2007.

[4] E. Balland, P. Brauner, R. Kopetz, P-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In F. Baader, editor, *Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[5] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, pages 301–317. Springer-Verlag, 2006.

[6] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In *Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming, ESOP 2007*, number 4421 in Lecture Notes in Computer Science, pages 125–140. Springer-Verlag, 2007.

[7] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *European Symposium On Research In Computer Security*, Lecture Notes in Computer Science. Springer-Verlag, September 2007.

[8] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer-Verlag, 2005.

[9] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *Asian Programming Languages and Systems Symposium*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, 2005.

[10] R. Bornat, P. W. O'Hearn, C. Calcagno, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages*, pages 259–270. ACM Press, 2005.

[11] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003.

[12] Y. Cheon and A. Perumandla. Specifying and checking method call sequences of Java programs. *Software Quality Journal*, 15(1):7–25, 2007.

[13] D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.

[14] P. Czarnik and A. Schubert. Extending operational semantics of the java bytecode. In *Trustworthy Global Computing: Revised Selected Papers from the Third Symposium TGC 2007*, number 4912 in Lecture Notes in Computer Science, pages 57–72. Springer-Verlag, 2008.

[15] J. evík. *Program Transformations in Weak Memory Models.* PhD thesis, University of Edinburgh, 2008.

[16] J. evík and D. Aspinall. On validity of program transformations in the Java memory model. In *European Conference on Object-Oriented Programming*, pages 27–51, 2008.

[17] Jaroslav evík. The Sun Hotspot JVM does not conform with the Java memory model. Technical Report EDI-INF-RR-1252, School of Informatics, University of Edinburgh, 2008.

[18] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Programming Languages Design and Implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.

[19] R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in c programs with recursive data structures. In K. Koskimies, editor, *International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, 1998.

[20] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In Z. Shao, editor, *Asian Programming Languages and Systems Symposium*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer-Verlag, 2007.

[21] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25(13):1741–1783, 1999.

[22] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, volume 5356 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, December 2008.

[23] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. Technical Report ICIS-R08014, Radboud University Nijmegen, 2008.

[24] C. Haack and C. Hurlin. Resource usage protocols for iterators. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, 2008. Revised version available from `http://www.cs.ru.nl/~chaack/papers/iterators.pdf`.

[25] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, July 2008.

[26] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. Technical Report 6430, INRIA, January 2008.

[27] C. Haack and C. Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 8(4):55–83, 2009.

[28] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1:35–47, 1990.

[29] M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle.* PhD thesis, Computing Science Institute, University of Nijmegen, 2001.

[30] M. Huisman and G. Petri. The Java memory model: a formal explanation. In *VAMP 2007: Proceedings of the 1st International Workshop on Verification and Analysis of Multi-Threaded Java-Like Programs*, number ICIS-R07021 in Technical Report. Radboud University Nijmegen, 2007.

[31] M. Huisman and G. Petri. BicolanoMT: a formalization of multi-threaded Java at bytecode level. In *BYTECODE*, ENTCS. Elsevier, 2008.

[32] C. Hurlin. Automatic parallelization and optimization of programs by proof rewriting. In *Static Analysis Symposium*, volume 5673 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, 2009.

[33] C. Hurlin. Specifying and checking protocols of multithreaded classes. In *Symposium on Applied Computing*, pages 587–592. ACM Press, March 2009.

[34] B. Jacobs. *A Statically Verificable Programming Model for Concurrent Object-Oriented Programs.* PhD thesis, Katholieke Universiteit Leuven, 2007.

[35] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.

[36] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[37] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer-Verlag, 2009.

[38] A. Lochbihler. Type safe nondeterminism - a formal semantics of Java threads. In *ACM Workshop on Foundations of Object-Oriented Languages*, 2008.

[39] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Principles of Programming Languages*, pages 378–391, 2005.

[40] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from `http://mobius.inria.fr`.

[41] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Principles of Programming Languages*, pages 327–338, New York, NY, USA, 2007. ACM Press.

[42] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Programming Languages Design and Implementation*, pages 308–319. ACM Press, 2006.

[43] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[44] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[45] M. Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, 2005.

[46] D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inria.fr/bicolano`. Summary appears in [40], 2006.

[47] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In Giuseppe Castagna, editor, *European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2009.

[48] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[49] M. Steffen. Object-connectivity and observability for class-based object-oriented languages, 2006. Habilitation thesis.

[50] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.