Mobius

Information Society Technologies

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 D3.3

Preliminary report on thread-modular verification

Due date of deliverable: 2007-03-01 (T0+18)

Actual submission date: 2007-03-28

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

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

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, 6
RUN	1, 3, 4, 5, 6
UCD	1, 2, 6

Executive Summary:
Preliminary report on thread-modular verification

This document summarises deliverable D3.3 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 intermediate results of Task 3.3 on the verification of multithreaded applications. Existing logic-based approaches to the verification of multithreaded programs do not scale to realistic examples, because they require to inspect or explicitly specify all possible interleaving points. Instead, in this task, we aim at identifying conditions that allow ignoring the interference of other threads on the thread and method that we wish to verify. If we do not have to consider possible interference, we can re-use well-known techniques for sequential program verification.

The document is structured as follows. Chapter 2 describes the impact of the Java Memory Model on the work in this task. The Java Memory Model describes the set of legal executions of a multithreaded application. An important objective for the Java Memory model has been to ensure that if a program does not contain data races, its set of legal executions should be sequentially consistent, i.e., it can be described by an interleaving semantics. However, if a program does contain data races, compiler optimisations can have unexpected results and the set of legal executions can become too complex to grasp for a human being. Indeed, data races are generally considered bugs, and thus applications are supposed to be free of data races. Therefore, we have decided that in the scope of the MOBIUS project we first check whether a program is free of data races, and only apply further verification techniques if this is the case. To support this decision, we have formalised the Java Memory Model in Coq, building on the semantics developed earlier in Task 3.1, and formally proved that all executions of data race free programs are sequentially consistent. Moreover, we have revived the RCC tool, which is a static checker that can be used to detect race conditions. Applying RCC to check for data race freeness will thus always be the first step in the verification process of an application.

Chapter 3 then identifies several conditions for thread-modular verification, i.e., conditions that ensure that methods are immune to interference of other threads, namely contract-atomicity, immutability, and thread ownership. Contract-atomicity is a generalisation of the classical notion of atomicity, where a method is atomic if it contains at most one instruction that is sensitive to interference. We extend this definition to take a method's specification, or contract, into account, thus ensuring that the validity of the method's contract cannot be affected by another thread. We sketch a verification method for contract-atomicity.

Immutability of an object guarantees that there is no need to synchronise accesses to the object. We propose a set of rules and a type system that allows us to identify whether an object is immutable, even in the presence of malicious code.

For thread ownership we propose annotations to indicate how many threads can access a object simultaneously, combined with so-called locking policies. The thread ownership system is very flexible, in particular because it supports transfer of ownership. If we know that an access to an object is always protected by a lock, or that there is at most one thread at the time that can access an object, then we know that other threads cannot interfere. We sketch a verification method for our thread ownership annotation system.

Chapter 4 proposes an extension of the Java Modeling Language with a set of keywords that are specific to multithreaded applications. These keywords are partly based on the results presented in the earlier chapters, and partly an improvement of an earlier proposal by Rodríguez et al.

Chapter 5 sketches two example verifications to illustrate how our different conditions for thread-modular verification can be combined in practice. The first example uses thread ownership and contract atomicity to verify a typical worker-tread pattern application. The second example uses immutability and contract-atomicity to verify an instance of the copy-on-write pattern. Both examples are based on patterns from Doug Lea's book on concurrent programming in Java.

Chapter 6 gives a brief summary of the results and presents the plans for the remainder of this task. Our plans are divided in two lists. The first list presents the topics that we plan to handle within the course of the MOBIUS project. The second lists contains topics we consider to be interesting for further study, but are not sure to have the time and possibility to deal with within the context of the MOBIUS project.

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 [38, 41, 68]. For multithreaded applications some initial theoretical investigations have been made into their verification [2, 3, 65], but so far these results have not led to any accepted practical logic-based verification method.

This report describes the first results of our investigations on how to develop a practical and sound verification method for multithreaded applications. In particular, we have studied how to specify typical program properties, such as atomicity [51, 30], immutability [35] and thread ownership [10], that help divorce the multithreaded aspects from the functional aspects of the specifications. Exploiting this separation of concerns allows us to use well-known sequential verification techniques to verify the functional behaviour of a single thread. In addition, we have also studied the role of the Java Memory Model [53] which specifies all possible executions of a multithreaded application. After reviewing these program properties (atomicity, immutability, and thread ownership) and the JMM, we summarise the methods and tools that are being developed in the course of the MOBIUS project, and we demonstrate the verification process of security requirements for multithreaded applications on a small set of concrete examples.

1.1 The Role of the Java Memory Model

First, Chapter 2 describes the role of the Java Memory Model (JMM). In order to show that an application respects a security requirement, one has to show that all possible executions of the application respect the requirement. The JMM specifies the legal executions of an application. In order to allow common compiler optimisations, statement re-orderings are allowed; therefore, the set of legal executions in general cannot be described by an interleaving semantics. However, the JMM provides a strong guarantee: for all applications that are *correctly synchronised*, the set of legal executions only contains *sequential consistent executions*, i.e., executions described by an interleaving semantics. A program is said to be correctly synchronised if it does not contain any data races, i.e., there cannot be simultaneous accesses to the same variable where at least one of the two accesses is a write action. Programs that are not correctly synchronised often exhibit unexpected behaviour and are difficult (or impossible) to verify. Consequently, since most data races are generally considered bugs, we first verify whether a program is free of data races, and develop further verification techniques only for correctly synchronised programs.

Concretely, this means that we are studying the following issues. To formally establish the guarantee for correctly synchronised programs, we have formalised the JMM and proven the guarantee in Coq. We have also made an extension of the Bicolano JVM semantics developed in Task 3.1 (see *http://mobius.inria.fr/bicolano*) with an interleaving semantics for multiple threads. This BicolanoMT semantics will be connected with the executions of a correctly synchronised program permitted by the JMM. Finally, we have implemented a race condition checker (based on the RCC checker [27, 1]) that checks whether a program is correctly synchronised.

1.2 Conditions for Thread-modular Verification

Chapter 3 describes the conditions that we have identified for allowing thread-modular verification. A fundamental problem with the use of pre- and post-conditions in specifications for multithreaded code is the problem of *interference*. By this, we mean the possibility that upon entry or exit of method `m` in one thread, another thread may break the pre- or post-condition of `m` that the first thread establishes or assumes. Reasoning about code in a thread-modular way requires some way of restricting or ruling out interference. Different avenues for doing this are (*i*) relying on a locking discipline that rules out interference on some part of the state if certain locks are held; (*ii*) exploiting the fact that certain objects are immutable and therefore immune to interference; (*iii*) using some form of thread ownership that rules out interference on objects which are not accessible by other threads.

To express when a program does not suffer from interference, the literature proposes the notions of atomicity and independence. Various analyses have been proposed to check for atomicity and independence. However, these definitions and analyses typically do not take the method specifications into account. Whereas these notions guarantee that semantically we can consider certain methods as atomic, they do not support modular verification in terms of method contracts. Therefore, Section 3.1 proposes new notions of atomicity and independence: *contract-atomicity* and *contract-independence*. A method can be verified sequentially if the part of the method's code that is relevant to the method specification is atomic or independent. If a method is contract-atomic with respect to its contract, then the contract is effectively thread-safe, and thus we can rely on it irrespective of interference by other threads.

Sections 3.2 and 3.3 describe how the notions of immutability and thread ownership help to rule out interference (and thus to establish contract-atomicity), because both thread-local (i.e., owned by a single thread) and immutable objects cannot be interfered with. To allow the use of shared mutable objects that are not thread-local in pre- and post-conditions we have to rely on a locking discipline: if a thread owns the locks of the objects mentioned in pre- and post-condition then interference can be ruled out.

Finally, we would like to emphasise that immutability also allows objects to be shared between trusted and untrusted code. Indeed, APIs to trusted code typically accept and return immutable objects as arguments and results. We have precisely formalised the notion of immutable object, and developed a technique to formally guarantee immutability, even in the presence of malicious code.

1.3 Specifications of Multithreaded Applications

Chapter 4 describes our proposal for an extension of the Java Modeling Language (JML) [48] with constructs that describe specific multithreaded aspects of an application. This extension language embodies the conditions for thread-modular verification identified in Chapter 3, but also supports the description of other typical design decisions related to multithreaded applications that may be useful for verification. The specification language that we propose is based on an earlier proposal by Rodríguez et al. [62], but corrects several problems and omissions of the original proposal. Also, the semantics of our language is more precise, and we are currently working on describing the semantics formally. This formalisation will be used to prove the soundness of our verification techniques.

Chapter 5 illustrates the use of our specification language and (thread-modular) verification techniques on some non-trivial examples. The examples are adaptations of typical coding patterns for multithreaded applications, as can be found in Doug Lea's concurrency package [47].

Finally, this document is an intermediate report describing the progress of MOBIUS Task 3.3 (Verification of multithreaded Applications) after 12 months. This initial period has mainly been used to investigate the different directions of research. Chapter 6 describes the plans for the remaining period of the task.

Chapter 2

The Role of the Java Memory Model

The Java Memory Model [44] (JMM) defines all legal executions of a multithreaded Java program. After giving a brief overview of the area of memory models, this chapter describes our formalisation of the JMM, which we use to prove some fundamental requirements of the model. In particular we prove that any correctly synchronised program is sequentially consistent. To provide a formal basis for this, we have extended the Bicolano [60] semantics with multiple threads and interleaving semantics. To complete the verification chain, we have developed (revived) a tool to check whether a program is correctly synchronised. This tool is described in Section 2.4 of this chapter.

2.1 Memory Models for Multithreaded Applications

With the emergence of multiprocessor architectures, shared memory has shown to be a simple and comfortable communication model for parallel programming. However, this simple abstraction requires for synchronisation mechanisms to keep the memory of the overall system up-to-date in the presence of concurrent accesses to shared memory locations. These synchronisation mechanisms can have a strong impact on the performance of the system. To overcome this problem, several relaxations of the consistency (or coherence) of the memory system have been proposed [4, **?**]. These relaxations will affect the programmability (i.e., the way programmers reason about their programs) of the overall architecture, since unexpected behaviours can result from them. In general, the more performance the memory system provides, the harder it is to reason about the programs running in that environment. A common example where this trade-off is evident is the use of write-back caches in a multiprocessor architecture; this can greatly leverage the write latency, but not all the processors will be able to see values written to variables stored in the cache, until the consistency protocol requires the write to be committed to the main memory.

A *memory model* defines all the possible outcomes of a multithreaded program running on a shared memory architecture that implements the model. In essence, it is a specification of the possible values that read accesses on the memory are allowed to return[1]. Therefore, it specifies the multithreaded semantics of the platform.

Since the memory model describes the allowed executions, having a precise specification of it is fundamental to guarantee the correctness of multithreaded programs. It also serves as a contract for both the programmer and the provider of the platform; for the programmer, it states which are the guarantees that can be assumed from the platform; for the provider, it defines which are the minimal conditions that it must comply with—which in turn has an important impact on the compiler and hardware optimisations the system can take advantage of. In the case of a high level programming language as Java, the memory model restricts which are the optimisations that can be applied by the source code to bytecode compiler and by the bytecode to native code compiler (e.g., a JIT compiler), and it also specifies which extra operations

[1] We will in general—in conformance with the memory model terminology—talk about the write that a read sees, instead of the write that deposited in the memory the value a read returned, as in general we do not care about the value, but about the write action itself.

must be inserted (in case it is needed) by the Java Virtual Machine (JVM) implementation to preserve the semantics mandated by the model on any particular architecture. The absence of a precise memory model can lead to serious security issues.

The basic building blocks of the memory model specification are single accesses to the shared memory. The specification indicates the *atomicity*, *visibility* and *ordering* conditions for these accesses, as perceived by the different processors; where atomicity defines which are the accesses that execute in a single indivisible operation, visibility defines which actions a processor can observe when they are performed by another processor, and ordering gives the restrictions on the order in which different processors might see the actions.

A wide range of memory models exist in the literature; Higham et al. [**?**] present a good introduction, definitions and the most important results in the area; Adve et al. [4] present a survey on memory models and a framework to classify them according to the restrictions and guarantees these provide.

The simplest memory model is *sequential consistency* [46] proposed by Lamport. The definition of sequential consistency states that the following condition can be assumed for a sequentially consistent program:

> "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

An important consequence from the above definition is that for every sequentially consistent execution, there should exist a *total order* over the memory actions, that has to be consistent with the *program order* (the order of the actions as dictated by the sequence of instructions in the program text), and where each read access sees the previous write to the same location according to that order. An interesting remark is that operations do not need to be actually executed in *that* total order, but can be overlapping or reordered; the definition only requires the result of the execution to be *as if* the actions were executed in total order and one at a time. The importance of the existence of that order is that it describes an *interleaving* of the memory actions of the different threads of execution. This allows to verify the program by simply considering all the possible interleavings of the actions; this is widely known as *interleaving semantics* for concurrency. Therefore, sequential consistency is attractive from a programmer's point of view. However its simplicity comes at a high cost on the performance of the system, because it disallows many uniprocessor compiler optimisations, and in addition, in some architectures memory barriers or other synchronisation mechanisms must be inserted to keep the memory consistent.

Uniprocessor compiler optimisations are widely used by sequential code compilers, and have been long studied and implemented. Performing multiprocessor compiler optimisations (which involve interactions of several threads) is a hard task for a compiler, because it has to analyse statically all possible outcomes of the program under the modifications that it wants to apply and to guarantee that no new outcomes are allowed due to these modifications. Therefore, most multiprocessor compiler optimisations are just uniprocessor optimisations applied to one of the threads, but as we shall see, performing uniprocessor optimisations can easily break sequential consistency. Nevertheless, for performance reasons we want to allow as many uniprocessor compiler optimisations as possible.

To allow for further performance improvements through optimisations, a consensus—mainly in the industry—was established that *weaker* restrictions than those of sequential consistency should be allowed for memory models. This raised a wide variety of new models which emerged from different relaxations of sequential consistency, in general called *weak memory models*. As emblematic examples we mention *weak ordering* [25], *processor consistency* [33], *release consistency* [32], among many others. Later, Adve et al. and Gharachorloo et al. defined the *data race free* [5] and *property labelled* [32] memory models, respectively, which take a different approach; namely, they require the programmer to identify actions involved in critical sections to guarantee the correctness of the program. The weak ordering and data race free memory models are of special interest to understand the Java memory model in the following section.

A *data race* occurs when an update on a memory location occurs concurrently with another access to the same location (read or write) by a different thread. *Weak ordering* was the first model to distinguish memory operations in *synchronisation* and *data* operations. In general, operations involved in data races

`x == y == 0`

Thread 1	Thread 2
`r1 := x;`	`r2 := y;`
`y := 1;`	`x := 1;`

Result: `r1 == r2 == 1`?

Figure 2.1: Unexpected behaviour

should as a rule be synchronisation operations, while common accesses (not involved in data races) should be data operations. synchronisation operations are guaranteed to have sequentially consistent semantics among them, while data operations can be reordered (a result of single-threaded compiler optimisation).

Data race free memory models take a different approach; they provide a minimal guarantee to the programmer, if he adheres to certain rules; optimisations are allowed provided that they do not violate these guarantees. The fundamental guarantee they build upon is *Data Race Freeness* (DRF):

> Correctly synchronised programs have sequentially consistent semantics.

A program is said to be correctly synchronised if when executed in a sequentially consistent manner no data races can appear in any of its executions. More precisely:

> A program is *correctly synchronised* if all its sequentially consistent executions are free of data races.

Determining what constitutes a data race in a sequentially consistent execution depends on the synchronisation mechanisms provided, because these prevent concurrent accesses to the same location to happen (recall that a data race occurs when to accesses to the memory location, such that one is a write, can happen at the same time). For that purpose the programmer, as previously, has to identify synchronisation and data operations, and based on that information the compiler or the architecture is allowed to reorder them. However in the presence of data races unexpected behaviours might happen[2].

Figure 2.1[3] shows an example of a possible unexpected behaviour, due to a common uniprocessor compiler optimisation. In the example variable names that start with r are registers local to the thread and are used to represent read actions, so for example `r1 := x` means a read action of the variable `x`, write actions are just assignments to the shared variables (which in general we will denote as `x`, `y`, `z`). In a sequentially consistent memory environment the only possible results for this program are: (*i*) `r1 == r2 == 0`, (*ii*) `r1 == 1 & r2 == 0` or (*iii*) `r1 == 0 & r2 ==1`. A common optimisation re-orders *independent* actions on different memory locations. This is allowed in uniprocessors, since the semantics of the program does not change if the instructions are independent of each other. However, in a multiprocessor environment this reordering can be perceived by another processor accessing the locations of the instructions being reordered. In this case, if any instruction in either Thread 1, Thread 2, or both are reordered the result `r1 == r2 == 1` could be a possible result of the program. For a common programmer this can appear to be an incorrect result, but it is actually allowed by most weak memory models.

In the previous example, accesses to both variables `x` and `y` can be involved in data races, since there is no synchronisation to prevent a read to happen while the write in the other thread is happening and vice versa. Most programmers aim for data race free code, since data races can cause unpredictable behaviours [56]. Verifying programs that contain data races is a hard task. Fortunately, in general the presence of races in multithreaded programs is an evidence of a bug. For the verification techniques described in the following chapters we will assume that programs are free of data races (correctly synchronised). Section 2.4 describes a tool that we use to check data race freeness and reject programs that contain races in some of its sequential executions.

[2]Unexpected in this context refers to the point of view of the programmer.

[3]Many of the examples correspond to examples in The Java Memory Model [53], this one corresponds to Figure 2.

However, notice that sometimes data races are not bugs, and are indeed intended. Benign data races, such as the unsynchronised assignment of the hash code of an object, are a typical example of this. In such cases it is important that the memory model guarantees that the program has certain restricted behaviour, and that a data race cannot be exploited to introduce security threats. We will describe the Java memory model in the next section, and we will point out how these requirements are met by the specification.

2.2 The Java Memory Model

The Java Memory Model specification [44, 53, 52] has been recently replaced, as the original was fatally flawed [61]. This specification defines the semantics for multithreaded Java programs. Therefore it serves as a guide for Java programmers, and as a specification for JVM and Java compiler implementors.

The requirements for the specification emerged from the consensus of a team that included researchers involved in memory models and concurrency, compiler and JVM constructors and expert programmers. The formal definition of the memory model is stated as a set of rules that every execution must satisfy and proofs are given to show that the rules meet the requirements.

2.2.1 Requirements and Motivations

One of the main motivations to replace the old JMM was that it did not allow most of the common single-threaded compiler optimisations. Therefore producing efficient multithreaded bytecode was almost impossible. For that reason, an important requirement to fix the model was to allow as many single-threaded optimisations as possible while keeping the programmability of the language fairly simple. A weak memory model emerged as a natural choice for this requirement.

Data Race Freeness As in many weak memory models, the programmability is guaranteed by requiring data race free programs only to exhibit sequentially consistent behaviours. In the context of the JMM (and memory models in general) this property is called the Data Race Freeness (DRF) property, as mentioned above. Another way to state the DRF property is:

> If every sequentially consistent execution of a program is free of data races, then these are all the possible executions, i.e., only sequentially consistent executions are allowed.

Note that the DRF guarantee allows to verify correctly synchronised programs by just analysing the possible interleavings of the threads, and without having to take into account possible re-orderings or other compiler optimisations.

Interestingly, the following property was identified by the authors of the JMM as a requirement to ensure DRF: no *Out of Thin Air* (OoTA) reads should happen. Furthermore, they identified that OoTA is also a desirable property in the presence of data races to avoid security problems. The OoTA property will be discussed below.

There are two means to achieve data race free Java programs through the use of synchronisation; namely, monitors and volatile variables. The basic actions on monitors are *lock* and *unlock*, and its semantics is that at most one thread can hold the lock of a monitor at a time. As expected, every lock action on a monitor is matched with a following unlock on the same monitor; moreover, when a lock is acquired by a thread, all the memory updates of threads that previously issued an unlock on that monitor are visible to the former thread. In case of volatile variables, the semantics defines that every access to them is atomic, and a read on a volatile makes all the memory updates of threads that executed a write on the same variable previously, visible to the reading thread (as in the case of the lock action). We will show later how these concepts are guaranteed by the JMM specification.

`x == y == 0`

Thread 1	Thread 2
`r1 := x;`	`r2 := y;`
`if (r1 == 1)`	`if (r2 == 1)`
`  y := 1;`	`  x := 1;`
	`if (r2 == 0)`
	`  x := 1;`

Result: `r1 == r2 == 1`?

Figure 2.2: Dependency breaking

Reordering As mentioned above, an important motivation for the redefinition of the JMM was to allow single-threaded optimisations. Many common compiler and hardware optimisations can be seen for simplicity as a reordering of instructions [53]. The *reordering of adjacent independent instructions* is probably the most basic optimisation that could be required. For that purpose, a clear notion of *independence* is needed; operations are independent if they execute on different memory locations. This does not mean that any two independent operations can be reordered, there are certain conditions that limit this requirement—for example instructions cannot be reordered outside synchronisation blocks to which they belonged originally—but in general the rule applies to every independent operation. An example of the application of this reordering rule has been presented in Figure 2.1.

A more delicate compiler optimisation that was required to be allowed by the JMM, involves the elimination of apparent dependencies in the code. When a static analysis of the code can detect that a write action is guaranteed to happen (i.e., the same value written to the same variable by the same thread) in every execution, the compiler is allowed to move that write earlier, thus removing the dependency; this can lead to really unexpected behaviours. Figure 2.2[4] presents an example where the write of `1` to the variable `x` is guaranteed to occur in any sequentially consistent execution, since the only possible values for the `y` variable are `0` and `1`. Therefore the write to `x` will always happen. A compiler doing this kind of reasoning can decide to eliminate the `if` conditions and reorder the read of `y` and the write of `x` in thread 2 (note that these actions are on different memory locations), and thus, the result `r1 == r2 == 1` is valid for the program. We refer the curious reader to the JMM definition [52, 53] for more details and examples on this reordering rule.

Removing *redundant synchronisation* and *synchronisation coarsening* [24] can produce important improvements in the performance of a program. We do give here the details of these optimisations and their implications on the JMM, but is important to know that allowing these is also a requirement for the model.

Volatiles Volatile variables are required to have atomic semantics, in other words, there should be a total order among them in the execution such that every volatile read sees the last write on the same variable in that order. Furthermore, it is required that volatile actions act as synchronisation points among the threads issuing the accesses. More precisely, when a read observes (via a volatile read) the value another thread deposited in the memory (with a volatile write), all the updates previous to the write made by the writing thread are immediately (and necessarily) visible to the reader thread. Therefore, volatile variables can be used as a synchronisation mechanism to guarantee that the effects of one thread are perceived by other threads.

2.2.2 Specification

This subsection summarises in a concise way the main ideas, and the formalism of the model. However, this is not a complete presentation of the model, for the complete definition the interested reader should consult the Java Memory Model specification [53].

[4] Figure 7 in The Java Memory Model [53]

`x == y == 0`, `m` is a monitor

Thread1	Thread2
1: `lock m;`	5: `lock m;`
2: `x := 1;`	6: `y := 1;`
3: `r1 := y;`	7: `r2 := x;`
4: `unlock m;`	8: `unlock m;`
po: $1 \xrightarrow{po} 2 \xrightarrow{po} 3 \xrightarrow{po} 4$, $5 \xrightarrow{po} 6 \xrightarrow{po} 7 \xrightarrow{po} 8$	
so: $1 \xrightarrow{so} 4 \xrightarrow{so} 5 \xrightarrow{so} 8$	
sw: $4 \xrightarrow{sw} 5$	
hb: $1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4 \xrightarrow{hb} 5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8$	

(a) Exec: Thread1;Thread2;

`x == y == 0`, `m` is a monitor

Thread1	Thread2
1: `lock m;`	5: `lock m;`
2: `x := 1;`	6: `y := 1;`
3: `r1 := y;`	7: `r2 := x;`
4: `unlock m;`	8: `unlock m;`
po: $1 \xrightarrow{po} 2 \xrightarrow{po} 3 \xrightarrow{po} 4$, $5 \xrightarrow{po} 6 \xrightarrow{po} 7 \xrightarrow{po} 8$	
so: $5 \xrightarrow{so} 8 \xrightarrow{so} 1 \xrightarrow{so} 4$	
sw: $8 \xrightarrow{sw} 1$	
hb: $5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8 \xrightarrow{hb} 1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4$	

(b) Exec: Thread2;Thread1;

Figure 2.3: Ordering examples.

First, we would like to clarify some standard concepts that might not be clear for readers not familiar with memory model terminology.

Programs Programs are, as usual, a set of instruction sequences (each element of this set being a thread) which are presented in the examples as simplified bytecode instructions with a syntax biased to make memory operations evident.

Executions In the context of the JMM an execution consists of the set of *actions* that it performs, *orders* over those actions that define how these actions are issued, and of course the functions that define for each read, which is the write that stored in the memory the value returned by that read.

Actions Actions are single accesses to the memory, they are classified in: reads or writes of ordinary variables, reads or writes of volatile variables (synchronisation actions), locks and unlocks on monitors (synchronisation actions), and some other actions that we will not enumerate here as they are not fundamental for the understanding of the model. Some actions are marked as external actions: these are used to define the observable behaviour of an execution.

In order to satisfy the DRF requirement we need a concrete definition of the concept of data races in the model. This is reflected in the JMM by the *Happens-Before* (`hb`) order, defined by Lamport [45]. The definition of `hb` captures visibility requirements, that is, when an action in a thread is *obliged* to see the effects of updates on the memory issued by another thread. To be able to define this order we first define three other simpler orders on actions, namely *Program-Order* (`po`), *synchronisation-Order* (`so`) and *synchronises-With-Order* (`sw`).

Program order The program order is a partial order that relates actions in the same thread. It basically represents the ordering of the instructions as given in the program text. This means that the actions of a single thread are totally ordered by `po`. Also note that actions on different threads cannot be related by `po`.

As an example of the orderings defined in this section look at Figure 2.3 where the actions have been numbered to facilitate the representation of the orders. We will take the execution in Figure 2.3(a) as a running example, the other possible execution is depicted in Figure 2.3(b). In our example we have chosen the execution that completely executes the thread Thread1, and then executes the Thread2 (recall that these definitions are concerned with particular executions and not with the program).

synchronisation order As said before, the JMM required synchronisation actions to execute sequentially consistent with each other. By the definition of sequential consistency there should exists a single

total order over the synchronisation actions of the execution, this order is called in the JMM the synchronisation order `so`.

In the previous example, `so` only involves the lock and unlock actions (the only synchronisation actions in the example). The order represented in the figure corresponds to the same execution as in the previous example.

The synchronisation order is needed to guarantee that `hb` is a partial order.

synchronises-With order The `sw` order can be derived from the `so` order. It is simply a *directed per variable restriction of* `so`. By directed we mean that only volatile writes and unlocks can appear in the first component of the ordered pairs, and only volatile reads and locks can appear in the second component of the pairs respectively. By restriction we mean that every pair that relates actions on the same variable (or monitor) present in the `so`, such that the first element is a lock and the second is an unlock, or the first is a volatile write and the second a volatile read, must appear in the `sw` order. We call the elements in the first component of the pairs *releases*, and the elements in the second component *acquires*.

In the example of Figure 2.3(a), the only edge from a release to an acquire is the one from action 4 to action 5, and in Figure 2.3(b) this edge goes from action 8 to action 1.

Happens-Before order Finally, the `hb` order specifies which are the writes (therefore, the values written) a read action is allowed to see (or return). This order is the transitive closure of the union of the `po` and `sw` orders $[(\texttt{po} + \texttt{sw})^*]$. As we said, `hb` formalises the conditions under which a thread is forced to see memory updates of other threads, so it is important to define which are the writes a read is able to observe. In simple words the `hb` edges generated by `po` pairs define that a thread must be aware of all its previous updates to the memory. The `sw` edges relate parts of the code of different threads such that every action that a thread could see before a release must be seen by every action after an acquire.

The definition of `hb` allows us to give a first approximation to the JMM, but it does not cover all of our requirements; in particular it is not sufficient to guarantee the DRF property. This approximation is based on the `hb` relation to define the visibility and ordering restrictions of the actions. The authors of the JMM [53] called this model the *Happens-Before Memory Model* (HBMM).

Basically the `hb` relation allows to specify *uniquely* which write sees each read in a data race free execution. For executions that contain data races the `hb` relation allows a read to see a write on the same variable that immediately precedes it in the `hb` order, or it is not `hb` related to it (a *data race*). This leaves a lot of freedom for implementors of compiler optimisations to choose according to their needs which is the write to be seen by those reads.

In what follows we show what a Happens-Before memory model for Java would look like. The reason to look at this model is that it is a good approximation to the kind of executions we want to allow (and reciprocally to disallow).

Happens-Before Memory Model In a HB memory model the values a read can return[5] are given by:

- a volatile read is only allowed to read the value written by the immediately preceding volatile write to the same variable in the `so`;
- `so` is consistent with mutual exclusion, i.e., no two locks by different threads occur on the same monitor before an unlock; and lock and unlock actions must be properly nested in `so`;
- an ordinary read is only allowed to see the immediately preceding write in `hb` on the same variable *or* a write on that variable that is not `hb` related to it (this last case constitutes a *data race*).

`x == v == 0`, `v` is volatile

Thread 1	Thread 2
`x := 1;`	`r1 := v;`
`v := 1;`	`r2 := x;`

Figure 2.4: HB Example

`x == y == 0`

Thread 1	Thread 2
`lock m`	`lock m`
`r1 := x;`	`r2 := y;`
`y := r1;`	`x := r2;`
`unlock m`	`unlock m`

Figure 2.5: Example where HB is enough.

The example in Figure 2.4 shows how this model restricts executions. The important fact of this example is that the variable `v` is volatile; therefore if the thread 2 reads the write in thread 1 on variable `v` a `sw` link relates the actions in both threads and thus the read of `x` in thread 2 is obliged to see the write of `x` in thread 1. In case the read in thread 1 does not read a value of `1`, the read in thread 2 can still see the write of `x` in thread 1, but in this case it forms a data race. Finally, the execution where thread 2 happens entirely before thread 1 is allowed.

We show now how HBMM in general guarantees the DRF property through the use of synchronisation (though it is important to know that it does not cover all the cases). Figure 2.5 depicts a correctly synchronised program (recall that correctly synchronised means that every sequentially consistent execution is free of data races). In any execution of this program, by the second bullet of the conditions of the HBMM there must be a `sw` link from the unlock of the first thread to the second one. This guarantees that there must be a `hb` link from any read to the write that it sees[6], therefore no data races can occur, and the HBMM conditions are enough to guarantee DRF in this example.

However, the HBMM is not enough, Figure 2.6[7] shows an example of a correctly synchronised program where a non sequentially consistent execution is allowed by the HBMM. In this example, every sequentially consistent execution only contains the two read actions, because in any interleaving of the threads both reads see a value of `0`, and therefore the writes are not executed.

We could imagine a compiler that optimises code by performing speculative execution of actions and if it can find an execution where that actions are justified it could decide that the speculation is valid, and therefore, the code can be optimised.

For example, in Figure 2.6, without loss of generality assume (speculatively) that the write `y := 42` in Thread1 happened; then, the read `r2 := y` in Thread2 is allowed to see that write returning the value `42`; this, in turn, validates the guard `(r2 != 0)` and justifies the write `x := 42` on Thread2; finally, the read of `x` in Thread1 is allowed to see the write on the variable `x` and return the value `42`; therefore, the guard in Thread1 would also be valid, justifying the whole execution, and in particular the execution of the initial speculative write of `y`. Notice that the execution above can only be justified using this kind of *causal* reasoning.

The result above is obviously not possible in any sequentially consistent execution, and the program is correctly synchronised, so to guarantee DRF the JMM must disallow it. But, as we can see the `hb` order does not prevent it from happening, since there are no `hb` links between the reads and the writes on the different threads. It is important to keep in mind for the following sections that if it was not for the data

[5] In general we can consider locks and unlocks as read and write actions that have further requirements such as the nesting and locking semantics.

[6] Default writes of variables happen before any access to the same variable.

[7] Figure 4 in The Java Memory Model [53]

x == y == 0

Thread 1	Thread 2

r1 := x;
if (r1 != 0)
  y := 42;
	r2 := y;
	if (r2 != 0)
	  x := 42;

Figure 2.6: Example where HB is not enough.

x == y == z == 0

Thread 1	Thread 2

r1 := y;
x := 1;
z := r1;
	r2 := x;
	y := 1;
	r3 := z;
	if (r3 + r2 == 2)
	  z := 42;

Figure 2.7: Justifying through data races.

races (non hb ordered accesses) the execution depicted could not have happened.

Any of the reads of 42 in the example is what the authors of the JMM call an *Out of Thin Air* (OoTA) read. The precise concept of OoTA is hard to state as we shall see. The main idea is that a read can *only* see values that could actually have been written by some execution where the write that stores that value is not involved in a circular causal justification through data races (as in the example).

Causality Requirements

This subsection gives an introduction to the problems around the definition of OoTA, and why it is hard to state it clearly and formally.

The example in Figure 2.6 showed that we must disallow OoTA values to guarantee DRF. But we also would like to guarantee that no value that could not have been written will ever be read, even in the case of executions that contain data races. The authors of the JMM tried to restrict the behaviour for every program (and not only for the correctly synchronised ones), to avoid security threats in case of data races.

A key idea is to identify when a write action can occur. As we saw in the example of Figure 2.6, the problem appeared when we assumed that a write that does not happen in any sequentially consistent execution could have occurred. Thus a first approach could be to forbid writes that do not appear in any sequentially consistent execution, and, by extension, to allow writes that occur in some sequentially consistent execution. This is not enough; as we can see in Figure 2.7, the write of 42 to z would never happen in any sequentially consistent execution. But if we consider that r1 and r2 could see a value of 1 (as we saw in the example on Figure 2.1), then r3 + r2 could evaluate to 2, and thus, the write should be allowed (it should be justified).

The example shows that we need a way to define which actions are allowed based on the actions that we know for certain that are already allowed. This suggests an *iterative procedure* to validate (or *commit*) actions, where based on the actions already committed, more actions are committed until all actions are justified. This raises the question of when an action can be committed. A good approach would be to commit an action if it is present in an execution where all the previously committed actions are executed, and all actions not committed execute in a sequentially consistent manner.

In Figure 2.8[8] it can be observed that sequential consistency of the actions not committed is not enough to avoid the behaviour shown. In particular the value 42 could be written to x in some sequentially consistent executions, allowing us to commit the reads r1 and r2 with that value. Then, we can commit the read of r0 seeing a different value; namely, the default write to z. This is clearly a violation of the OoTA property, since

[8]Figure 12 in The Java Memory Model [53]

`x == y == z == 0`

Thread 1	Thread 2	Thread 3	Thread 4
`r1 := x;` `y := r1;`	`r2 := y;` `x := r2;`	`z := 42;`	`r0 := z;` `x := r0;`
Result: `r0 == 0` & `r1 == r2 == 42`?			

Figure 2.8: Thread 3 does not write 42.

`r1` and `r2` see values that no thread ever wrote. There are some other issues with the previous definition which we will not discuss here but are more precisely defined in the JMM [53]. The main idea to restrict OoTA reads, is that the justification of the value `42` occurs only when we have a data race—between the actions on Thread 3 and Thread 4—that is used to justify them. So our intuition to define the OoTA is that when an action is being committed all the actions not committed cannot see a value through a data race; thus preventing the behaviour depicted in the Figure 2.8. In the JMM such executions are called *well-formed* executions.

Note that there is much more discussion about the OoTA property in the JMM [53], and it is still a topic under discussion[9]. Here we only to give an intuition to the reader, we do not pretend to give complete explanations and motivations for the rules of the model.

The Java Memory Model Formally

Now we turn attention to the formal definition of the JMM. This is the mathematical definition of the model; thus to check if a Java multithreaded execution is allowed we must see that it fits the requirements stated below.

First we define more precisely the concepts introduced in the previous section and then we give the causality requirements formally.

Actions An action a is a tuple, with the shape $< t, k, v, u >$, where

- t stands for the Thread Identifier of the thread issuing the action;
- k is the kind of the action, which could be: *read*, *write*, *volatile read*, *volatile write*, *lock* or *unlock*;
- v stands for the variable or monitor on which the action executes; and
- u is interpreted as an *Unique Identifier*, which distinguishes every action from any other action.

Executions An execution E is represented as a tuple $< P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow} >$ where:

- P is the program that generated E.
- A is the set of all actions executed in E.
- $\overset{po}{\rightarrow}$ is the Program Order (as stated before) of E.
- $\overset{so}{\rightarrow}$ is the Synchronisation Order (as stated before) of E.
- W is the Write-Seen function that for each read in A assigns a write to the same variable also in A.
- V is the Value-Written function that for each write in A assigns a *value*.
- $\overset{sw}{\rightarrow}$ is the Synchronises-With order (as stated before) of E.
- $\overset{hb}{\rightarrow}$ is the Happens-Before order (as stated before) of E.

[9] http://www.decadentplace.org.uk/cgi-bin/mailman/listinfo/cpp-threads

It is important to note that in this document we only present the basic definitions and elements to specify the memory model, but we made some simplifications. For example, some kind of actions are missing in the previous definitions, nevertheless the missing kinds are not crucial for the understanding of the model.

In order to be able to formally define well-formedness of executions we need to introduce formal definitions of the `sw`, the `hb` and the `ssw` orders.

synchronises-With In the JMM the first component of a `sw` pair is called a *release* and the second component is called an *acquire*.

There are `sw` edges from[10]:

- an unlock of a monitor to every lock on the same monitor following it in `so`;
- a volatile write on a volatile variable to every read on that variable following it in `so`; and
- the default write of each variable to the first action in every thread.

Happens-Before As mentioned before the `hb` order is obtained by the transitive closure of the union of the `sw` and `po` orders. Mathematically: $hb = (po + sw)^*$.

Sufficient synchronisation (`ssw`) `ssw` are the `sw` present in the transitive reduction of the `hb` order. This is the minimal set which is included in `sw`, such that when we take it in conjunction with the `po` order and apply the transitive closure of the union we obtain the `hb` relation as result. In Manson's thesis [52] this set is claimed, but not proved, to be unique.

Well-Formed Executions

The definition of well-formed executions gives some basic formal requirements to the executions of the JMM. This is a lower bound upon which we construct the more complicated rules of the causality requirements. Every execution used to justify actions in the justification procedure must be well-formed among other requirements.

An execution $E =< P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow} >$ *is Well-Formed* if:

1. **Read write consistency** Each read of a variable x sees a write to x. An action x is volatile iff $x.v$ is a volatile variable.

   Mathematically[11]:

   $$(\forall r \in Reads(A) : W(r) \in A \wedge W(r).v = r.v) \wedge$$

   $$(\forall a \in A : Volatile(r.v) \iff a \in VolatilesReads(A) \vee a \in VolatilesWrites(A))$$

2. **synchronisation Order has an order less or equal to omega** This implies that for every action in the `so` order there are only a finite number of actions that occur before it.

3. **Consistency of `so` with `po`** synchronisation Order is consistent with Program Order and Mutual Exclusion. This consistency requirement implies that `hb` is a partial order. Mutual exclusion guarantees the locking semantics (one at a time), and proper nesting of locking.

   Mathematically: (Only the consistency property).

   $$\forall x, y \in A : x \overset{po}{\rightarrow} y \wedge y \overset{so}{\rightarrow} x \Rightarrow x = y$$

[10] There are some other `sw` edges that we do not consider here as they are not fundamental.

[11] We use $Reads(A)$ to denote the set of read actions in A, similarly for $Writes(A)$, $VolatileReads(A)$, $VolatileWrites(A)$, $Locks(A)$, $Unlocks(A)$. The predicate $Volatile$ is valid only if its parameter is a volatile variable.

4. **Intra-thread consistency** The actions *performed* by every thread of the execution are the same that would be generated by the thread in isolation provided that each read r reads the value $V(W(r))$ and each write w writes $V(w)$. The `po` given must reflect the order in which the actions would be performed according to the Java specification, without taking the JMM into account.

5. **Consistency on `so`** For every volatile read r in A it does not happen that $r \overset{so}{\rightarrow} W(r)$. And, for every w, a volatile write action on $r.v$, it does not happen that $W(r) \overset{so}{\rightarrow} w \overset{so}{\rightarrow} r$.

   Mathematically:

$$(\forall r \in VolatileReads(A) : \neg r \overset{so}{\rightarrow} W(r)) \wedge$$
$$\neg(\exists w \in VolatileWrites(A) : w \neq W(r) \wedge r.v = w.v \wedge W(r) \overset{so}{\rightarrow} w \overset{so}{\rightarrow} r)$$

6. **Consistency on `hb`** For every read r in A it does not happen that $r \overset{hb}{\rightarrow} W(r)$. And, for every w, a write action on $r.v$, it does not happen that $W(r) \overset{hb}{\rightarrow} w \overset{hb}{\rightarrow} r$.

   Mathematically:

$$(\forall r \in Reads(A) : \neg r \overset{hb}{\rightarrow} W(r)) \wedge$$
$$\neg(\exists w \in Writes(A) : w \neq W(r) \wedge r.v = w.v \wedge W(r) \overset{hb}{\rightarrow} w \overset{hb}{\rightarrow} r)$$

Causality Formally

The well-formed execution rules above are general rules, specifying simple and expected conditions on executions; but they do not state anything about about data races, or the justification procedure mentioned before. Restrictions on the presence of data races are expressed by the causality rules. In particular, these rules specify which actions can be committed in each step of the justification procedure, and thus, how the presence of OoTA reads is avoided.

The justifying procedure starts with an empty set of committed actions, and in each step one or more actions are added to the commitment set. When an action is committed, it must remain committed until the end of the procedure, when all actions of the execution must be justified. For that purpose we define pairs containing commitment sets and justifying executions, one for each iteration of the procedure, where commitment sets are strictly increasing through the justification, and the justifying executions are well-formed executions that must meet the causality rules to commit further actions as we will see below.

A justification of a well-formed execution $E =< P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow} >$ is is a sequence of pairs (C_i, E_i), where C_i represents the i^{th} commitment set and E_i is the i^{th} commitment execution. Note that the sequence can be infinite if the actions in the execution are infinite. All justifying executions are well-formed, and only actions of the execution being justified (also present in the justifying execution) are committed.

The following conditions must hold for the commitment sets in the sequence:

- $C_0 = 0$
- $C_i \subset C_{i+1}$
- $A = \bigcup(C_0, C_1, C_2, ...)$

A simple way to understand the commitment procedure is to think that we want to find an execution E_i of the program P that imitates E when restricted to the committed actions, and where the actions we are *now* trying to commit happen independently from the data races not yet committed, i.e., we forbid not committed data races in justifying executions.

The following rules state that the actions committed (justified) have to be present in every execution that follows its commitment in the sequence, and that the `hb` and `so` orders of the justifying and justified executions have to coincide on the committed actions.

1. $C_i \subseteq A_i$

2. $\overset{hb_i}{\rightarrow}|_{C_i} = \overset{hb}{\rightarrow}|_{C_i}$

3. $\overset{so_i}{\rightarrow}|_{C_i} = \overset{so}{\rightarrow}|_{C_i}$

The next rule only refers to write actions, (i.e., the domain of the V function), and says that when a write is committed the value it writes to has to be the same as the value it writes to in the justified execution. Another way to say that is that the value written is permanent when committed.

4. $V_i|_{C_i} = V|_{C_i}$

The rule below is important. Analogously to the previous case, it only refers to reads (i.e., the domain of the W function). The rule says that when a read is being committed or it is not committed yet, it can see in the justifying execution (E_i) a different write than the one it sees in the justified execution (E); but furthermore, it says that when a read r is being committed (i.e., $r \in C_i - C_{i-1}$) must see the final write (i.e., W_r) in the the next commitment. We will see how this rule allows to avoid the justification of actions with data races when combined with the rules 2, 6 and 7, while it allows to commit (valid) data races.

5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

Rule 6, is also important. It tells us, as outlined before, that reads in the justifying execution that are not committed can only see writes that are related by `hb` with them, i.e., no data races are allowed for actions not already committed.

6. $\forall r \in Reads(A_i - C_{i-1}) : W_i(r) \overset{hb}{\rightarrow} r$

The next rule specifies that when a read is committed, both, the write that it sees in the justifying execution (E_i), and the write that it sees in the original execution E must be committed.

We can understand this rule as the requirement we stated before, the actions that justify a read must have happened (i.e., must be committed), in this case the write is in E_i; but as per rule 6 when a read is justified it can only see a write that happens before it, this would not allow data races to be committed. That is the reason why rule 5 talks about the previous commitment set, and it is why this rule talks about the write in the execution E as possibly being different from the write the reads sees when its being committed. We will see that in an example soon.

7. $\forall r \in Reads(C_i - C_{i-1}) : W_i(r) \in C_{i-1} \wedge W(r) \in C_{i-1}$

Rule 8 is related with synchronisation actions. It says that whenever a synchronisation link is used to commit an action, that link must remain there. Recall that synchronisation links extend the `hb` relation, and thus, can allow some reads to see writes that are not still committed, provided that the write happens before the read. So removing synchronisation links would allow to commit reads that see a write that is yet not committed and it could be not related by `hb` to the read on forthcoming justifications if we are allowed to remove the link. A possible more operational intuition would be, that if an action is justified assuming that a synchronisation of the memory was issued between two threads, we must guarantee that the synchronisation actually occurred at that point.

8. $\forall x, y, z \in A_i : x \overset{ssw_i}{\rightarrow} y \overset{hb_i}{\rightarrow} z \Rightarrow (\forall j \geq i : x \overset{sw_j}{\rightarrow} y)$

Finally, rule 9 requires all external actions that are ordered before any committed action by `hb` to be committed too. This rule is of importance only to define the observable behaviour of executions, but not to prove the proposed requirements of the model.

9. $\forall x, y \in A_i : External^{12}(x) \wedge y \in C_i \wedge x \overset{hb_i}{\rightarrow} y \Rightarrow x \in C_i$

`x == y == 0`

Thread 1	Thread 2
`r1 := x;`	`r2 := y;`
`y := r1;`	`x := r2;`

Figure 2.9: Causality rules example

Figure 2.9 presents an example that could generate OoTA reads if the compiler did speculative execution and the JMM would not prevent it. For that reason we will use it to show how the causality rules prevent OoTA reads from happening. The result in question is `r1 == r2 == 42`[13], note that this result is allowed by the HBMM.

As in the result depicted both reads see a value of `42` both writes must store a `42` in the memory. Without loss of generality we will only show which are the conditions that are needed to commit the read of the variable `x`. If those conditions cannot be met, that read is not possible. Per rule 7, when a read r is committed, both, the write that sees in the justifying execution ($W_i(r)$) and the write that sees in the final execution ($W(r)$) must be already committed. The only candidate to write a value of `42` to the variable `x` is the write on Thread 2, so this write must be committed before the read in Thread 1, and it must be committed writing that value, per rule 4. But to respect intra-thread semantics, required by the well-formedness of justifying execution, the read of `y` in Thread 2 must see a value of `42`, because the following write stores a `42`, though that read can only see the default write to the `y` variable since it is not committed and rule 6 demands not committed reads to see a write ordered before them by $\overset{hb_i}{\rightarrow}$. Would we have started trying to commit the other write, a similar reasoning would forbid it (from the symmetry of the program). Therefore a write of `42` can never be committed, invalidating the intended result.

Furthermore, the only result possible for that program is `r1 == r2 == 0`.

The example in Figure 2.6 is even simpler. As we mentioned, to be able to justify the execution we had to assume that a write of `42` was issued first (and thus has to be committed first). There are no well-formed executions of this program where the reads see writes that happen before them, and where the any of the guards evaluates to true, therefore no write of `42` can be ever committed.

2.2.3 Proof of Data Race Freeness

Two important proofs sketches are given in the JMM paper [53]; namely that the specification allows reordering of adjacent statements and the most important in terms of verification of multithreaded applications, that the DRF property holds. Given its importance for the MOBIUS project we give a rough sketch of the DRF proof.

The DRF guarantee requires every execution of a correctly synchronised program to be sequentially consistent. Recall that sequentially consistent executions are those for which a total order on its actions can be found such that every read sees the last write previous to it on the same variable in that order; and correctly synchronised programs are those where every sequentially consistent execution is free of data races (i.e., no read sees a write not ordered before it by `hb`).

The proof goes in two steps. First, a lemma is proved that saying that whenever in an execution of a correctly synchronised program all reads see writes that happen before them, the execution is sequentially consistent. Then the actual DRF property is proved using the lemma and proving that in every execution of a correctly synchronised program reads see writes that happens before them. The proofs can be encountered in the JMM paper [53].

Lemma 2.2.1. *For every execution E of a correctly synchronised program P; if every read sees a write that happens before it, E is a sequentially consistent execution.*

[12] The definition of external actions is the expected one.

[13] Note that the value 42 is arbitrary, just to show that any value (not written) could happen.

The main theorem proof shows that every read of an execution of a correctly sees a write that happens before it, therefore, by Lemma 2.2.1 the execution must be sequentially consistent.

Theorem 2.2.2. *Every execution E of a correctly synchronised program P is sequentially consistent.*

An issue may be to explore is a proper formal definition for the OoTA property and a proof that the JMM satisfies that property. This would allow for a better understanding of how the rules interact with each other and also could show deficiencies on them, since now it is hard to reason about all the possible executions of the JMM.

Our approach to the verification of multithreaded software within the MOBIUS project it is to verify only correctly synchronised programs. As mentioned before, this decision is based on the fact that in general data races are evidences of bugs in the program. Even though there are programs that contain data races which are intended by the programmer, reasoning and verifying those programs is really hard, and it is out of the scope of the project. Therefore, we are mainly concerned with the DRF guarantee of the JMM which allows us to use interleaving semantics. We rely on techniques to detect data races to reject incorrectly synchronised programs as will be discussed in Section 2.4.

As we strongly rely on the DRF guarantee, we formalised the JMM in the proof assistant Coq [20], and we proved it mechanically to have certainty of its correctness. Some parts of the specification are not currently formalised as these are not needed to prove DRF. In particular, we did not formalise yet the definition of observable actions and behaviour, or rule 9 that restricts observable actions. Further, we also did not formalise yet finalisers and the different guarantees for constructors (such as dereference chains, safe contexts etc.).

To be able to work with interleaving semantics we extended the Bicolano [60] formalisation in Coq of the Java Bytecode semantics with multiple threads of execution, this will be developed in Section 2.3.2.

2.2.4 Formalisation in Coq

To be able to prove the DRF guarantee mechanically we formalised the JMM as literally as possible, closely following the paper by Manson et al. [53]. With this we expect to avoid problems derived from misunderstandings or biased interpretations of the rules. As we mentioned above, there is no need to formalise the whole JMM to prove DRF, hence, there are some parts which we did not formalise in the current version of the model, though these can be easily extended afterwards.

Here we give a list of the main restrictions (and for some, we indicate how we could extend the model to overcome them).

Finite executions For the moment we only consider finite executions. There are several reasons that justify this approach. First, infinite executions could generate infinite justification sequences (in the sense of the justification process of the JMM). This raises several issues that need to be carefully handled, for example the requirement that the union of the sets of committed actions must converge to the set of all actions in the execution must be given as a coinductive predicate. To consider infinite justification sequences coinductive data structures should have been used, and this would make the formalisation more complicated, hence we left it for future work. Another reason is that we need to talk about the transitive reduction of the `hb` relation with possibly infinite elements. This is unique for a finite quantity of element but we cannot take this for granted for infinite executions. We have not worked on a proof of the uniqueness of the transitive reduction of the `hb` in the infinite case, and it is not clear that is a fundamental requirement of the model. However, as infinite executions are not a priority for this version of the formalisation we leave this analysis for later work.

Kind of Actions Some actions, for example thread creation and thread termination, are not covered in the current version of the model. This is because these actions are not fundamental for the specification, or to prove the DRF guarantee, and can be easily added afterwards. However these actions are not included just for simplicity reasons, there is no theoretical problem to add them to the formalisation, so it is also left as future work.

Sequential Continuation Axiom For the proof of DRF we need to show that given a prefix of an execution there is a sequentially consistent execution that has that prefix. To be able to prove this we would need to establish a link between the program instructions and the actions of the JMM. The basic idea would be to have a sequentially consistent scheduler that chooses an instruction, executes it and then proceeds infinitely or until the program finishes. At this time we have not linked the formal definition of the model with the semantics of single-threaded Java, so we cannot talk about program instructions in the current formalisation. This is something we expect to do, but for this first version of the model we take it as an assumption.

Topological Sort of a Partial Order A topological sorting of the `hb` order is needed to prove the DRF property. Constructing all the possible topological sorts of a partial order is more easily achieved if we are able to see the partial order as a forest of directed acyclic graphs (DAG). This would require transforming the representations we currently have for partial orders, which are sets of ordered pairs, to DAGs. For the moment we give an axiomatic specification of a the topological sorting, and leave the implementation of the algorithm on graphs for later study.

First occurrence Axiom Also for the DRF proof we need to say that given a total order derived from the topological sort of the `hb` there is a first occurrence of a property that relates the elements in the order (first with respect to the order given). We added this first occurrence as an axiom. Proving it instead of assuming it would require to know that every topological sort of the `hb` order has an order type less than or equal to omega (i.e., there are only a finite quantity of actions preceding every action in the order). From the definitions of well-formed executions we know that `so` has an order type less than or equal to omega, with this assumption the same property about the `hb` should be provable, and then the same property should be provable for any topological sort of the `hb`. We have left these issues for future work and assumed it for now.

In this section we make a more specific description of the formalisation in Coq of the JMM. We will briefly describe some of the main definitions and axioms as well as the important proofs we give. We would like to clarify that not all definitions here are needed for the DRF proof, but some are added just for completeness and to be able to easily extend the model afterwards.

Building Blocks Some data types such as thread identifiers (`Thread_ID`), unique identifiers for actions (`UID`), variable names (`VarName`) and values (`Value`) are not of great importance in the specification, therefore we formalised them as abstract data types augmented with a decidability predicate, that allows us to compare elements of the same type.

Below we depict the definition of the Kind data type, which models the kind of actions existing in the JMM. It was formalised as an enumerated data type. We have covered only the kind of actions that have a real impact on the proofs we wanted to give, as we described before. One little difference with the specification of the JMM is that the volatility condition of some of the actions is captured by a synchronisation function instead of the kind type.

```
Inductive Kind :  Set :=
         | Read | Write
         | Lock | Unlock
         | VolatileRead | VolatileRead.
```

Actions & Executions Actions and executions are formalised axiomatically using the Coq module system. Below we present a fragment of the Actions module. We implemented Actions as an abstract data type which has several functions associated to it, like for example the `act_kind` function that given an action returns the kind of the action (read, write, etc.).

```
Module Type ACTIONS.
      Parameter t :  Set.
      Parameter act_kind :  t → Kind.
      Parameter act_thID : t → Thread_ID.t.
      Parameter act_uid :  t → Uid.t.
      Parameter act_var :  t → VarName.t.
      Parameter synchronisation :  t → bool.
      Parameter act_dec :  ∀ x y:t, {x = y} + {x ≠ y}.
End ACTIONS.
```

Executions are axiomatised in a similar way. The only parameters we need for executions are the program `P`, the set of actions `A`, the `so` order, the `po` order and the `W` and `V` function. The `sw`, the `hb` and the `ssw` orders are defined using the other orders as described above[14].

```
Module Type EXECUTIONS.
      Parameter t :  Type.
      Parameter program :  t → Program.
      Parameter actions :  t → Ensemble Act.
      Parameter po :  t → Relation Act.
      Parameter so :  t → Relation Act.
      Parameter W : ∀ (E:t)(x:Act), actions E x → act_kind x = Read → Act.
      Parameter V : ∀ (E:t)(x:Act), actions E x → act_kind x = Write → Val.
      Definition sw (E:t) :  Relation Act :=
               fun x y:Act => so E x y ∧ act_var x = act_var y ∧
                    act_kind x = Write ∧ act_kind y = Read.
      Definition hb (E:t) :  Relation Act :=
               clos_trans (fun x y:Act => sw E x y ∨ po E x y).
```

Some properties about the elements of the definition are given as axioms. As an example we state (axiomatically) that `po` is a partial order and that it is total when restricted to the actions of a single thread.

```
Parameter po_partial_order :  Order (po E).
Parameter po_total_upto_threadEq :
         ∀ t:Thread_ID.t, total_upto (po E) (fun x:Act ⇒ act_thID x = t).
```

The predicate `Order` is the conjunction of the predicates that say that a relation is reflexive, transitive and antisymmetric, as expected; and the predicate `total_upto` says that a relation is total when restricted to a certain set of actions, in this case the actions which belong to the same thread.

The fact that `hb` is a partial order is an interesting example of what can be proved only with the specification of these modules. This proof uses that `so` is a total order over the synchronisation actions, that `po` is total per thread and that `so` and `po` are consistent. The proofs of reflexivity and transitivity of `hb` are trivial since both `sw` and `po` are reflexive and, `hb` is the transitive closure of their union. The proof of the antisymmetry requires the conditions mentioned, but we do not present it here, since it is not of great interesting for the rest of the chapter and the details are somewhat cumbersome.

Justifications The justification procedure is axiomatised through a function that takes a program, an execution and a proof that the execution belongs to that program, and returns a list (with at least one element) of pairs containing an execution and a set of actions. The ordering in the list represents the ordering of commitments in the justification, and the elements in each pair are the execution used to commit actions, and the set of committed actions so far, respectively. The definition in Coq is:

[14]Here we do not give the formalisation of the `ssw` order, as it is complicated and not necessary understand the `Executions` module

```
Definition  Justification.t := Executions * Commitments.
Definition  commitE := sList Justification.t.
Parameter   justification :
            ∀ (P:Program)(E:Exec), eIn (Program_Executions P) E → commitE.
```

The `Justifications.t` definition represents the pairs contained in the sequence of justification, and `commitE` represents the sequence (`sList` is a library for non-empty lists). The `justification` parameter gives the type of the function that represents the committing procedure, in that expression `eIn` represents that execution `E` (not shown here, but declared in the scope of the definition) belongs to the set `(Program_Executions P)`; where the function `Program_Executions` returns, for a given program, the set of all its possible executions (recall that this is only an axiomatic definition). The other parameters given in this module specify how these executions and sets of committed actions are related. For example, the definition below states that each committed action must belong to the set of actions of the execution in the commit sequence ($\mathtt{E}_i$).

```
Definition  committedActionsInE :=
            ∀ (E_exec_P:eIn (Program_ExecutionsP) E)(j:Justification.t)(a:Act),
            In (justification E_exec_P) j →
            eIn (comm j) a →
            eIn (actions (exec j)) a.
```

Using the JMM notation the property above would be: $\forall j : C_j \subseteq A_j$.

Causality Rules The causality rules are stated as literally as possible. We have formalised all the rules, but actually some rules are not needed for the proofs we gave in this version of the model. We have added them just for completeness, and to be able to extend our formalisation to cover more properties as future work.

As an example of the formalisation of the rules, rule 2 is phrased in Coq as follows:

```
Definition  req2 :=
            ∀ (E_exec_P:eIn (Program_ExecutionsP) E)(j:Justification.t)(x y:Act),
            In (justification E_exec_P) j →
            eIn (comm j) x →
            eIn (comm j) y →
            hb E x y ↔ hb (exec j) x y.
```

This definition says that the execution being used to justify actions (`exec j`), and the execution whose actions we are trying to justify (`E`) have the same `hb` ordering when restricted to the current committed actions (`comm j`). All the other rules follow the same general format.

Proofs The proof that correctly synchronised programs have only sequentially consistent executions is given in our model following the proof sketch by the authors of the JMM paper, mentioned above.

The assumptions mentioned before are enough to prove the theorem. The current state of the formalisation has the lemma completely proved and all the elements needed for the proof of the theorem are stated. The final proof of the DRF theorem is not written yet, since we plan to modularise it in smaller lemmas for readability purposes.

2.3 Multithreaded Bicolano

Bicolano [60] is the formalisation in Coq of the Java bytecode semantics, developed as part of the Task 3.1 of the MOBIUS project. Bicolano follow the Java Virtual Machine Specification [50] but it is only concerned with the sequential semantics of Java bytecode.

For correctly synchronised programs we rely on interleaving semantics as our formal model. For that purpose we extended the sequential version of Bicolano with interleaving semantics. This is still work in progress, thus some of the features are not yet covered in the current formalisation.

This section shows how we augmented the notion of state present in Bicolano to model all the possible interleavings of instructions.

2.3.1 Bicolano

First we give a brief recapitulation of single-threaded Bicolano. The bytecode semantics is specified axiomatically, with the use of several data structures that model the inner workings of Java.

Data Structures To define the state of an execution we need to define the *heap*, the *callstack* and the current *frame*. In this version of the interleaving semantics in Bicolano (we shall call it BicolanoMT from now on) we do not consider the exceptional state, so in the following we do not mention it, but it is formalised in Bicolano and has to be taken in account for the final version of BicolanoMT.

Similarly to the axiomatisation of the JMM, the operations on the *heap*, *callstack* and *frame* are specified through axioms that show which is the expected output for the input given. As an example, below we give the signature and the specification of the *get* operation on the *heap*.

```
Parameter get :  t → AddressingMode → option value.
Parameter get_update_same :
          ∀ h am v, Compat h am → get(update h am v) am = Some v.
Parameter get_update_old :
          ∀ h am1 am2 v, am1 ≠ am2 → get(update h am1 v) am2 = get h am2.
```

In the first definition the type of the *get* operation is given. The `AddressingMode` parameter specifies whether the location we are trying to access is a static field, a dynamic field or an element in an array. The semantics of that function is that for a given heap and an *AddressingMode*, the *value* placed in that location of the heap is returned. In case there is no value in that location a special value (*None*) is returned.

The `get_update_same` parameter specifies the behaviour of the *get* operation after an *update* operation on the heap on an *AddressingMode* `am` with the *value* `v`. In short it says that an access to the heap on a location must return the last value updated for that location. Similarly `get_update_old` specifies that the value returned when the argument is not the location being updated is the same as we had before the `update`.

With the definition of the `heap`, and similar definitions of `Frame` which contains information about the method currently being executed (such as the operand stack, the local variables and the program counter) and `CallStack` which is a list of frames, we can specify the program `state`.

```
Module Type STATE.
      Inductive t :  Type :=
                | normal :  Heap.t → Frame.t → CallStack.t → t
```

Small Step Semantics The small step semantics is defined in Bicolano with a large inductive definition which has a case for each instruction of the bytecode language, and it specifies in each case how the state is transformed after performing such an instruction. In the example below we can see how this is implemented for the `nop` instruction which leaves the state unmodified, but incrementing the program counter by one.

```
| nop_step_ok :∀ h m pc pc' s l sf,
               instructionAt m pc = Some Nop →
               next m pc = Some pc' →
               step p (St h (Fr m pc s l) sf) (St h (Fr m pc' s l) sf).
```

We can see in the definition above that if the instruction at the current program counter is a `Nop`, and the next program counter is `pc'`, the state resulting after the execution of the instruction is identical to the original state but with the `pc` modified. We can think of `step` as the predicate that defines the operational single step semantics.

Now we will see how we can extend the definition of state to model interleaving semantics.

2.3.2 BicolanoMT

Data Structures The main difference between sequential Bicolano and the multithreading extension BicolanoMT is that we have to take into account the *local* state of each thread, i.e., every thread has its own current *Frame* and *CallStack*. The definition for thread is as follows:

```
Module Type THREAD.
      Inductive t :  Set := make :
                Frame.t →
                CallStack.t →
                ThreadState.t →
                WaitingForLock.t →
                WaitingForThread.t → t.
End THREAD.
```

The possible states of threads are specified in the `ThreadState` definition which is an enumerated data type that contains as values `runnable`, `blocked` and `waiting` [50]. `WaitingForLock` is formalised as a memory location that contains the lock that is blocking the current thread, and `WaitingForThread` contains the `ThreadID` of the thread being waited for, when trying to "join" a thread. These parameters should in fact be fields of the thread object allocated in the heap, but for simplicity we added them to the thread definition. We believe that there is no real problem with this definition, but the formalisation could always be changed.

Once the thread type is defined we need a way to handle all the threads together. For that purpose we defined `ThreadMap` to be a mapping of `ThreadID`s to the state of the `Thread`.

```
Module Type THREADMAP.
      Parameter t :  Set.
      Parameter get :  t → ThreadId.t → option Thread.t.
      Parameter get_update_new :  ∀ tmap x v, get (update tmap x v) x = Some v.
      Parameter  get_update_old :
                ∀ tmap x y v, x ≠ y → get (update tmap x v) y = get tmap y.
```

We can now easily define the state of multithreaded executions by replacing the current `Frame` and `CallStack` the by the `ThreadMap`.

```
Module Type MULTISTATE.
      Inductive t :  Set := make :  Heap.t → ThreadMap.t → t
```

Single Step (Interleaving) Semantics The definition of a step of execution in interleaving semantics (`mtstep`) makes use of the definition of `step` of execution in the singlethreaded semantics. In fact, a step in a multithreaded program would be any step in any of the threads that are currently `runnable` or some special multithreading instructions that are not present in the sequential semantics. The definition of the `mtstep` predicate that represents the valid state transformations in multithreaded Java is as follows:

```
Inductive mtstep (p:Program) :  MState.t → MState.t → Prop :=
| interleave_mtstep :  ∀ tmap tmap' tid h h' f f' cs cs',
  ThreadMap.get tmap tid = Some (Tr f cs ThreadState.runnable None None) →
  tmap' = ThreadMap.update tmap tid (Tr f' cs' ThreadState.runnable None None) →
  step p (St h f cs) (St h' f' cs') →
  mtstep p (MSt h tmap) (MSt h' tmap')
⋮
| monitorenter_nonblocking_mtstep_ok :  ∀ m pc pc' tmap tmap' tid loc s l cs h,
  instructionAt m pc = Some monitorenter →
  next m pc = Some pc' →
  ThreadMap.get tmap tid =
  Some (Tr (Fr m pc ((Ref loc)::s) l) cs ThreadState.runnable None None) →
  Heap.isLockableBy h loc tid →
  tmap' =
  ThreadMap.update tmap tid (Tr (Fr m pc' s l) cs ThreadState.runnable None None) →
  mtstep p (MSt h tmap) (MSt (Heap.lock h loc tid) tmap')
```

In the `interleave_mtstep` case of the definition above, the interleaving of `runnable` threads is formalised. More precisely, the definition says that a state transformation in the interleaving semantics corresponds to the execution of a *step* of a single `runnable` thread, or some multithreaded instruction. All the multithreaded instructions must be added to the `mtstep` predicate. An example of a multithreaded instruction is `monitorenter`. Here we only present the case where the `monitorenter` grants access to the thread requiring it (remember that the exceptional case is not covered in the current model). The `monitorenter_nonblocking_mtstep_ok` case states that if the chosen program counter points to a `monitorenter` instruction (`instructionAt m pc = Some monitorenter`), the chosen thread is `runnable` and can lock the `loc` reference on top of the frame's operand stack, then the lock is granted to that thread.

Native methods are called using the `Invokespecial` instruction. This was not formalised in Bicolano at the time of writing BicolanoMT. As native methods do not have a matching bytecode instruction, placeholders (e.g., `_native_notify`) were used to define their semantics.

As an example we show the native start instruction which causes a thread to begin executing. On the stack there should be a reference *loc* to the thread to be executed, then we lookup the run method of the class of the thread to be executed, the *new* program counter is set to the first instruction of this method and the threads becomes *runnable* with an empty callstack, an empty operand stack and a self-reference in the local variables (recall that we do not cover exceptions here).

```
native_start_mtstep_ok :
      ∀ tmap tmap' tmap'' tid tid' h m m' pc pc' pc'' loc s l l' cs cn cl bm',
      instructionAt m pc = Some _native_start →
      next m pc = Some pc' →
      ThreadMap.get tmap tid = Some (Tr (Fr m pc ((Ref loc)::s) l) cs
            ThreadState.runnable None None) →
      ThreadMap.get tmap tid' = None →
      tmap' = ThreadMap.update tmap tid (Tr (Fr m pc' s l) cs
            ThreadState.runnable None None) →
```

```
Heap.typeof h loc = Some (Heap.LocationObject cn) →
lookup p cn (RunMethodSignature p cn) (pair cl m') →
METHOD.body m' = Some bm' →
pc'' = BYTECODEMETHOD.firstAddress bm' →

l' = stack2localvar ((Ref loc)::s) 1 →
tmap'' = ThreadMap.update tmap' loc (Tr (Fr m' pc'' OperandStack.empty l') nil
      ThreadState.runnable None None) →
mtstep p (MSt h tmap) (MSt h tmap'')
```

As said before the current version of BicolanoMT does not cover the exceptional state and there are some multithreaded instructions missing. We expect to complete all the instructions and support exceptions in future versions of the semantics.

We also plan to explore how to link the formalisation of the JMM with the formalisation of the interleaving semantics. In fact we are only interested in linking the part of the JMM that deals with correctly synchronised programs. A simplification of the JMM and other alternatives are being evaluated to achieve this goal.

2.4 A Tool for Race Detection

As it was presented earlier, the JMM offers programmers the following guarantee: *All possible executions are sequentially consistent if all sequentially consistent executions are free of data races.* To support reasoning about sequentially consistent concurrent systems, we must be able to identify when a program is free of data races. Thus, we have "revived" Race Condition Checker (RCC), a tool that certifies that a program is free of data races. After successfully using RCC other components of the MOBIUS verification environment may assume that all executions are sequentially consistent.

Recalling the following guarantees offered by the JMM is sufficient to understand the rest of the section:

- There can be no data race between actions of the same thread—actions ordered by the program are also ordered by happens before.
- There can be no data race on volatile fields—accesses to volatile fields are always ordered by happens before.
- Releasing a lock in one thread always happens before its subsequent acquire by another thread.
- All other conflicting memory accesses are data races.

2.4.1 Rules and Annotations

RCC was originally developed as a prototype by Flanagan and Freund [27] before Java's memory model was revised and formalised [44]. Abadi, Flanagan, and Freund have given a thorough presentation of the tool's theoretical underpinnings in a recent publication [1]. Here we present, informally, the rules imposed by the tool, what we call RCC 1.0, today.

Java type declarations are *thread shared* or *thread local*. All reference fields of a thread shared type must be thread shared. The RCC version we ship considers a type to be thread shared by default.

The fields of a thread shared type must be `final`, `volatile`, or `guarded_by` by a lock (a Java `Object`). Whenever a variable is accessed, all its protecting locks must be held. To avoid doing a must-alias analysis on lock variables, RCC forces locks to be `final` and simply compares them by name. Methods may *require* certain locks to be held whenever they are called. These are called the protecting locks of the method, and must be held whenever the method is called.

Type declarations are parametrised by *external locks*. These declarations permit programmers to name references that are not accessible at runtime. Whenever the type is used, lock arguments fulfilling the

```
/*#thread_shared*/ class Node/*#{ghost Object data_lock}*/ {
  public volatile Node/*#{data_lock}*/ next;
  public Object data/*#guarded_by data_lock*/;
}

/*#thread_shared*/ class LinkedList {
  private final Object x = new Object();
  private Node/*#{x}*/ head /*#guarded_by this*/;
  public void add(Object data) /*#requires x*/ {
    Node/*#{x}*/ n = new Node/*#{x}*/();
    n.data = data;
    synchronized (this) {
      n.next = head;
      head = n;
    }
  }
}

/*#thread_local*/ class User {
  public LinkedList list;
}
```

Figure 2.10: Example RCC annotations

parameterisation must be provided. Two parametrised types are the same if their Java types are the same and their arguments are identical. Analogous rules decide whether types are convertible, subtypes of each other, and so on.

All annotations are written in comments that begin with `/*#` and end with `*/` so that annotated code can be compiled as normal. The somewhat contrived example in Figure 2.10 illustrates all the available annotations. The words `ghost Object` need to be used when declaring formal external locks only as a result of a current implementation detail and should be ignored.

2.4.2 Example

Let us explore why the example in Figure 2.10 is correct.

The class `User` is thread local, therefore its field `list` need not be protected. The type of `list` is not given any arguments. This is correct because the declaration of `LinkedList` does not specify any formal external lock.

The declaration of `Node` does have a formal external lock and, as a result, whenever `Node` is used it is given an argument. In particular, it is also given an argument in the `new` expression that initialises the variable `n`. The variable `n` is later assigned to `head`. The assignment is type-correct because both `n` and `head` have the type `Node/*#{x}*/`. The field `head` can be accessed in this assignment because its protecting lock, `this`, is held.

The earlier access to `n.data` is more interesting. We are allowed to access `n` since it is a local variable. Its field `data` is protected by the external lock `data_lock`, which is, in fact, `x` because the type of `n` is `Node/*{x}*/`, and the lock `x` is held because it is required by the method `add`.

Finally, the field `data` is protected (by a lock) because it appears in a thread shared class (`Node`). The other field, `next`, is protected because it is marked as `volatile`.

Consider now the example in Figure 2.11, which illustrates a pattern that appears later on in Figure 5.1. A client of `OverThread` might think that it is safe to check if `isOver` is `true` before looking at the result.

```
class OverThread extends Thread {
  public boolean isOver = false;
  public Object result = null;
  public void run() {
    /* <Do something, not accessing isOver> */
    /* <Set result when finished> */
    isOver = true;
  }
}
```

Figure 2.11: A pattern to detect early when a thread finishes

But according to the JMM the statement that sets `isOver` can be moved to the beginning of the method `run`. RCC detects this problem in this example and forces us to either protect the variable with a lock or mark the variable as `volatile`.

2.4.3 Reviving RCC

When we started working on RCC, it did not even compile. Now it works in a Java 1.4 Virtual Machine and processes Java 1.4 code.

We present a few details of the architecture to understand why the code did not compile and why the changes we had to make were not just local. RCC shares a Java front-end (called "JavaFE" for short) with the Extended Static Checker for Java (ESC/Java) [19]. JavaFE is responsible for parsing and type checking Java and building an Abstract Syntax Tree (AST). RCC and ESC/Java are responsible for parsing their respective annotations and for doing extra type checks.

JavaFE is a large subsystem and it did not have any versioning scheme. Since RCC was originally developed, the API of JavaFE changed. Furthermore, RCC contained cut-and-pasted code from JavaFE and, in some cases, that code contained bugs that were later fixed, but only in JavaFE. The reason why the code was copied is because JavaFE was designed to be extended by subclassing but overriding methods did not provide the required granularity of control for RCCẆe have rewritten parts of JavaFE to permit a finer granularity of control and to eliminate this code duplication. In the process of rewriting parts of JavaFE, we have also fixed a number of bugs that affected ESC/Java.

The RCC test-suite has grown during this work as well. The testing framework now allows the developer to easily run a subset of tests. The results are presented either quietly or in a verbose mode, and all details are logged.

Additionally, RCC now understands and reasons about `volatile` variables. The old version would only allow us to fix the example in Figure 2.11 by protecting `isOver` with a lock. Now the field can be marked as volatile—a much more efficient solution.

From a theoretical point of view the most interesting aspect of this work was realising that type checking sometimes did not terminate in RCC unless extra restrictions were imposed on what can and cannot be used as an argument for an external lock. Figure 2.10 illustrates that we want to be able to use either normal fields or formal external locks. In general we might think of allowing any final expression, but this can be problematic.

Let us look closer at how we concluded that the access `n.data` in Figure 2.10 is permitted. The lock that protects `data` is `data_lock`. Because it is an external lock, we looked at the type of `n` and replaced `data_lock` with whatever was used as an argument. In this case the final expression used as an argument was `x` and we then saw that `x` is a lock we hold.

In general, locks are compared by names that are rewritten into a normal form before comparison. To better see this non-termination problem with type checking, we will use lowercase letters for normal (final) fields, uppercase letters for (formal) external locks, and we omit the use of the dot operator. Using this

```
class Type/*#{ghost Object A}*/ {
  Type/*#{A.A.a.a}*/ a /*#guarded_by a.a.A.A*/;
  void doit() { a = null; }
}
```

Figure 2.12: First example of non-terminating type checking

```
class Type/*#{ghost Object A}*/ {
  Type/*#{a.A}*/ a /*#guarded_by a.A*/;
  void doit() { a = null; }
}
```

Figure 2.13: Second example of non-terminating type checking

representation, a declaration of a variable `a` of a type that is parametrised by `A`, `B`, and `C` will introduce rewrite rules that transform `aA`, `aB`, and `aC` into whatever was used as arguments for the external locks.

This rewrite system `aA->AAaa` diverges when starting from `aaAA`. The corresponding Java example is seen in Figure 2.12.

A simpler rewrite system that does not terminate is `aA->aA`. The corresponding Java code is seen in Figure 2.13.

Both these examples do not make any sense. We need some additional rule to disallow them so that we do not go into an infinite loop while type checking the body of the method `doit`. The rule should be flexible enough to allow both uses that appear in Figure 2.10.

Expressing the problem in terms of a rewrite system makes it easy to find solutions. The one we chose to stick with is the following: an argument is either an external lock or a field access that involves only normal (final) fields. We explored other options and we believe that this one is flexible enough and offers readable error messages to the user.

Another approach would have been to stop type checking when a variable name grows above a certain limit. While this is practical, the only error message we could provide is something of the form of "lock 'l' is too complicated," and the user might not enjoy the fuzziness of that error message. Another approach would have been to try to impose an ordering on external locks and on fields and fail when this ordering seems impossible to find. Even if this approach supports use cases that we currently do not, it is more complex, and we still cannot give a comprehensible error message.

2.4.4 Next Steps for RCC

RCC has a number of known bugs that need to be addressed before it is used in an industrial setting.

First, we kept the RCC annotation syntax different than the Java Modeling Language (JML) annotation syntax. The RCC annotations have well defined semantics [1], but it is not clear what the semantics of combined annotations (with normal JML annotations) should be. For example, RCC external locks introduce new names. Can these names be safely used in another JML annotation? The following example shows a hypothetical (syntactical) way of integrating the annotations.

```
class C/*@{lock}*/ {
  //@ invariant \typeof(lock) == \typeof{LockClass};
}
```

Should this program be legal? Does it mean that, whenever `C` is instantiated, the lock provided should have runtime or static type `LockClass`? Should we use an invariant to specify such a property, or should we just rely on the type checking in RCC?

Numerous similar problems exist. In particular, integration with the Universes type system is nontrivial, but it suggests improvements [22, 9] that would make RCC more flexible. Other foreseeable benefits of such an integration include allowing RCC to be more flexible by processing the `immutable` keyword and including in JML keywords with well-defined semantics such as `guarded_by`.

In general, the remaining challenge is to make RCC accept most of well-understood, structured patterns programmers use when writing concurrent code while preserving soundness. This work will facilitate RCC's adoption and will make annotating legacy code easier.

Chapter 3

Conditions for Thread-modular Verification

Ideally, we would like to apply standard reasoning techniques for sequential code – using the established notions of method contracts – to multithreaded code. A central difficulty is that if we apply such techniques to verify a property of one thread, that property may be invalidated by *interference* of other threads.

Verification techniques such as Owicki-Gries and Rely-Guarantee support reasoning about threads in the presence of interference by other threads. In the Owicki-Gries approach the program code of other threads has to be known and non-interference has to be shown for any possible interleaving. This results in a highly non-modular verification technique [59]. The Rely-Guarantee approach achieves modularity by specifying restrictions on the interference by other threads that an individual thread assumes [43]. In our work on thread-modular verification techniques the aim is to somehow rule out interference by other threads so that we can apply standard techniques for reasoning about sequential code as much as possible. Similar techniques have been used successfully in other settings, e.g., Thornley's work on reasoning about concurrent C programs [66].

In this chapter, we describe our work in progress on statically verifying properties that help limiting thread interference. We have worked on identifying atomic code blocks (Section 3.1), identifying immutable heap regions (Section 3.2) and regulating object access (Section 3.3).

- **Identifying atomic code blocks.** A code block is atomic if for every (arbitrarily interleaved) program execution there exists a semantically equivalent execution that runs the block without interleavings from other threads. Well-designed multithreaded programs typically contain an abundance of atomic blocks. Knowing about atomic blocks is extremely helpful for program verification, because verification can soundly ignore all thread interleavings inside these blocks. We are working on extending the concept of atomicity and developing techniques for modularly verifying our new variant of atomicity (called *contract-atomicity*). The main idea is to define atomicity with respect to method contracts and take advantage of the constraints that contracts impose on the environment, in order to reach a more liberal notion of atomicity. Section 3.1 describes the main ideas.

- **Identifying immutable heap regions.** Immutable parts of the heap are only accessed by reads, but not by writes. Therefore read/write or write/write conflicts are impossible when accessing immutable regions. We are working on a type-based analysis for verifying object immutability. A type system for a core language has been designed and proven sound [35]. Section 3.2 describes the main ideas of this type system.

- **Thread ownership.** Access to a particular heap location cannot result in interference if, whenever one thread has permission to write to this location, no other thread has permission to access this location at all. We are working on *regulating object access* to ensure such an invariant for object access permissions. The goal is to leverage existing ideas [10] from a low level core language to an object-oriented setting. In this approach, object access permissions for threads may vary dynamically. This is

more liberal than policies based on thread-locality [27, 9, 42], where for each thread-local object there is only one thread that can access it and this thread remains fixed throughout the entire program run. This is work in progress. The main ideas are described in Section 3.3.

3.1 Exploiting Contracts for Atomicity

The notion of atomicity [31] (and the auxiliary notion of independence [62], coinciding with a sequence of both-movers in [51]) has been considered important for the verification of concurrent programs for some time.

Definition 3.1.1 (Atomicity)**.** *A code block is atomic if for every (arbitrarily interleaved) program execution there exists a semantically equivalent execution that runs the block without interleavings from other thread.*

If a method is atomic, we can assume that the method body executes without interleavings from other threads. Thus, we can significantly reduce the cost of program verification by reducing the number of interference freedom tests that program logics for concurrent programs require [59]. Earlier investigations have shown that typical concurrent programs contain an abundance of atomic methods [62], thus it is well worth exploiting this fact for thread-modular verification. However, existing approaches do not take specifications into account. We improve on these approaches by providing a notion of *contract-atomicity*, where atomicity of a method depends on the method's contract.

3.1.1 Previous Atomicity Analyses

Before presenting our new notion of contract-atomicity, we briefly review related work on this topic.

The literature contains a number of program analysis techniques for verifying atomicity. Most of these techniques are based on the *reducibility* criterion, as introduced by Lipton [51]. The reducibility criterion forms the basis for atomicity type systems [31, 30], dynamic atomicity analysers [29] and some model checking techniques for atomicity [36]. A method is said to be reducible if its implementation respects the pattern $R^*N^?L^*$ where R^* denotes zero or more so-called right-movers, $N^?$ denotes zero or one so-called non-mover and L^* denotes zero or more so-called left-movers. Any read or write to a variable that is protected by a lock is both a left- and a right-mover (also called a both-mover), a lock acquire is a right-mover, a lock release is a left-mover and any read or write to a non-protected variable is a non-mover. Lipton showed that, if a method respects this pattern, every concurrent execution can be reduced to a sequential one.

Later, Flanagan introduced a model checking technique that can verify the atomicity of certain code blocks that are not reducible [26]. This technique is based on executing serial and non-serial versions of an operational semantics simultaneously, and checking that both versions yield the same final state. Flanagan calls this model checking technique *commit-atomicity*, because it can verify a common atomicity pattern where a code block does not perform any action that affects other threads until a so-called commit point is reached. The following method, which makes use of an atomic compare-and-swap operation, matches this pattern:

```
void acquire() {
  boolean r := true;
  while (r==true) { CAS(m,false,r); }
}
```

This method is not reducible, because `CAS(m,false,r)` is not a mover. However, it is atomic as Flanagan's model checking technique shows.

In [28], Flanagan et al. present a type and effect system for verifying a semantic property they call *abstract atomicity*. Abstractly atomic code blocks are atomic with respect to an *abstract operational semantics*, which differs from a standard operational semantics by allowing additional non-determinism. Roughly speaking, the abstract operational semantics allows a skip-transition as an alternative for a read-transition (but not as an alternative for a write-transition). Methods that are atomic are also abstractly atomic, but the converse

```
int alloc(){
    int i = 0;
    int r = -1;

    while(i < max) {
        l[i].lock();
        if(free[i]) {
            free[i] = false;
            l[i].unlock();
            r = i;
            break;
        }
        l[i].unlock();
        i+=1;
    }
    return r;
}
```

Figure 3.1: Method `alloc`

is not true. Figure 3.1 shows an example of an abstractly atomic method that is not atomic. The `alloc`-method searches for a free disk block. The flag `free[i]` indicates whether the `i`th disk block is currently unused. This flag is protected by the mutually-exclusive lock `l[i]`. When `alloc` identifies a free block, it allocates this block by setting the appropriate bit to false and returns the index of that block, otherwise it returns -1. Although abstractly atomic, the method is not atomic w.r.t. a standard operational semantics, because there could be a concurrent thread that ensures there is always at least one free block, yet in an interleaved execution the search performed by `alloc` could still fail to find a free block. This is not possible in an execution that serialises `alloc`.

The following inclusion hierarchy summarises the relation between the different analysis techniques and semantic notions of atomicity that we encountered in the literature:

$$\text{reducibility [51, 31, 30, 29, 36]} \subset \text{commit-atomicity [26]} \subseteq \text{atomicity} \subset \text{abstract atomicity [28]}$$

3.1.2 Contract-atomicity

The traditional definition of atomicity is purely based on implementations. On the other hand, we assume that methods are annotated with behavioural contracts in the style of JML's multithreading extension [48]. This explicitness has an advantage over the previous approaches, because the contract helps us to understand what behaviours are acceptable for the method, and contract-atomicity let us know what the atomicity property means in terms of contracts.

For example, even if `alloc` is abstractly atomic, how does it simplify verification and what properties does the method have? Indeed, it is difficult to write a contract for `alloc` because the property the programmer would like to express is *"if `alloc` returns an index different from -1, the bit at this index has been set to false at some point in the past"*. Unfortunately, the current version of the JML language is not adequate to easily express such a property.

The following example (again, from [28]) illustrates the correspondence between the contract and the intended behaviour of a method. In the following code fragment, the value of the variable `packetCount` does not influence the correctness of the method `receive` and this is made explicit by the contract:

```
int packetCount;
Queue packets;
```

```
void enqueue(Packet p);

//@ ensures packets.has(p);
void receive(Packet p){
    packetCount++;
    enqueue(p);
}
```

By analysing where the contract (`ensures packets.has(p)`) reads when it evaluates, we can deduce that the intended behaviour of `receive` does not depend on `packetCount` (provided the call to `has` does not read it). Therefore, we propose a new notion of atomicity that takes the method specification into account:

Definition 3.1.2 (Contract-atomicity)**.** *A method m is said to be* contract-atomic *with respect to its contract c if, whenever c is respected in a sequential setting, c is also respected in a concurrent setting.*

Alternatively stated, interleavings from other threads do not change whether m respects c. An example of contract-atomic method is shown in Figure 3.2. In this example, the method `route` is contract-atomic because its post-condition cannot be violated even if the packet `p` passed as a parameter is shared between multiple threads. The method `route` is not abstractly atomic (thus, not atomic either).

As with previous notions of atomicity, contract-atomicity entails that the correctness in a sequential setting implies the correctness in a concurrent one. However, contract-atomicity differs from previous notions for two reasons. First, it requires that the intended behaviour of a method in a concurrent context is made explicit. Second, contract-atomicity has been designed to be modular. Indeed, on this latter point, previous definitions of atomicity suffer from being poorly modular, because the composition of two atomic regions may not yield an atomic region if each region contains a non-mover [62]. It means that if method m_1 contains a call to method m_2 which has already been declared atomic, one cannot use this result in a modular way: in order to verify the atomicity of m_1 it is mandatory to look at the implementation of m_2 again, for example by inlining it in m_1 when the analysis proceeds. The reason why contract-atomicity is modular is that it rules out possible external interferences (a similar approach has been taken in SCOOP [57]). An external interference occurs when the contract of a method is broken between the moment where the callee returns and the moment where the caller resumes. Typically, external interferences forbid to reason sequentially because they entail that contracts are not stable. Consider the following example where `otherMethod`'s post-condition is `R`:

```
//@ assignable x.f;
//@ requires P(x);
//@ ensures  Q(\result);
MyObject method(MyObject x){
  x.otherMethod();
                          // (1)
  return x;
}
```

When reasoning sequentially, one can assume that `R[this/x]` holds when `method` returns. We say that `R[this/x]` is *stable* [58] because it holds at least until the execution of the next statement. Yet, in a multithreaded environment, naively, no value is stable, because everything can always be changed by the environment. Respecting a locking discipline or using confinement permits to entail the stability of predicates or methods. An example at the end of the document illustrates this (see Section 5.1).

Thus, unstability forbids us to assume `R[this/x]` when `method` returns since other threads can modify fields of `x` at `(1)` and break this property. However, if `otherMethod` has been checked to be contract-atomic,

```
class BackupMachine {
    synchronized void add(Packet p) { .. };
}

class Packet {
    private Lock l;          // accesses to Packet objects are protected by the lock "l"
    private boolean treated;

    //@ requires l.isLocked();
    void treat(){ treated = true; }

    void protect() { l.lock(); }
    void isTreated() { return treated; }
}

class Router {
    Vector<Packet> treated; // accesses to this vector are protected by "this"
    BackupMachine backup;   // object shared between different threads

    //@ ensures p.isTreated();
    void route(Packet p){
        p.protect();
        p.treat();

        synchronized(this){
            treated.add(p);
        }

        backup.add(p);
    }
}
```

Figure 3.2: Example of a contract-atomic method: `route`

we have the property that the post-condition `R[this/x]` is stable until the next release of a lock (thus, at least until the next statement) and that is why it can be used when `method` returns. More generally, contract-atomicity permits to reason in a modular way because it forces contracts to be stable. Of course, it comes at a cost: in order for a method to be contract-atomic, the contract must not be sensible to external interferences. Indeed, a contract-atomic method m enforce callers to perform adequate synchronisation before calling m.

As mentioned above, the second point where contract-atomicity differs from previous notions of atomicity is that it requires the intended behaviour of a method in a concurrent context to be explicit and that is why contract-atomicity is not a strict extension of (abstract) atomicity: if a method is strangely specified, it may be reducible (as this does not consider the contract), but it may not be contract-atomic. For example, consider the following piece of code where the annotation `monitors_for x <- lock` indicates that accesses to `x` are protected by `lock` (see Chapter 4 for a complete description of such keywords):

```
Object x; /*@ monitors_for x <- lock @*/
Object y;

/*@ ensures P(x);
```

```
  @ ensures P(y);
  @*/
public void doSmtg(){
    lock.acquire();
    x = independent_call(x);
    lock.release();
}
```

Method `doSmtg` is reducible, because it respects the pattern $R^* N^? L^*$. However, it is not contract-atomic, because the contract asserts something about the field `y` and, as the implementation does not acquire any protection for this field, nothing can be ensured about it. Therefore, we define a notion of *well-formedness* of a contract which is used to warn the programmer that a method is likely not to be contract-atomic.

This notion is concerned with the set of locations read/written by a method and the set of locations read when its contract is evaluated. JML already provides multiple keywords for that purpose, e.g., `accessible` and `assignable`. The set of locations specified in `assignable` (resp. `accessible`) clauses is an upper-bound of the set of locations that may be written (resp. read) during the execution of a method.

Definition 3.1.3 (Footprint)**.**

1. *The* footprint *of a method is the union of the set of locations listed in its assignable clauses and the set of locations listed in its accessible clauses.*
2. *The* footprint *of a contract is an upper-bound of the set of locations read during its evaluation.*

The footprint of a method m only mentions objects visible by a client, i.e., it cannot mention objects that are local to m. In addition, the footprint is modular, because if m contains a call to method n, one can re-use n's footprint to compute m's footprint. In the following example, as `b` is an alias of `a` and it is is not visible to the client, m's footprint is `{a.x, a.y}`.

```
class Point {
  int x,y;

  //@ assignable y;
  void n(){ y = 3; }
}

//@ assignable a.x, a.y;
void m(Point a){
  Point b = a;
  b.x = 3;
  b.n();
}
```

Footprints of contracts are more difficult to compute than footprints of methods because of JML's expressiveness, in particular because of `\exists` and `\forall` expressions. If none of this keyword appears, the footprint of a contract is determined in a similar way as for a method (by using `accessible` clauses of methods called). If a quantifier is present, it is possible to compute the footprint of the contract considered if the domain of the quantifier is finite (this is the case if a `\forall` quantifier is used to range over all elements occurring in an array). That is why the footprint of `oneify`'s contract in the piece of code below is the whole array `a`.

```
//@ assignable a[0..a.length];
//@ ensures (\forall int i; i >= 0 && i < a.length; a[i] == 1);
```

```
void oneify(int[] a){
  for(int i = 0; i < a.length; i++)
    a[i] = 1;
}
```

If the domain of a quantifier cannot be determined, the worst case is assumed: the footprint of the contract considered is the whole heap. Note that we will change the semantics of JML visible-state invariants because, in its current state, it makes footprints useless. Indeed, according to this semantics, all invariants of all allocated objects are conjoined to non-helper methods[1] pre- and post-condition. Of course, this is not correct in a multithreaded program where a method may proceed whereas objects in the heap are not respecting their invariant at the same moment. We plan to use information about object ownership to provide a new notion of visible-state invariants.

Well-formedness is important, because often methods having badly-formed contracts are not contract-atomic. Thus, it can be used as a lightweight mechanism to warn the programmer for a potential specification error. Note that this definition implies that a contract whose footprint is the whole heap cannot be considered well-formed because its footprint is obviously bigger than the footprint of the method associated.

Definition 3.1.4 (Well-formed)**.** *A contract c is* well-formed *for a method m if the footprint of c is included in the footprint of m.*

Checking contract-atomicity. We are currently working on the development of an algorithm to check contract-atomicity. Its formal definition and soundness proof are future work; here we sketch the main ideas behind the algorithm. It will eventually be integrated into the MOBIUS tool set. The algorithm will rely on an alias analysis and read/write analysis and the notion of stability [58].

An alias analysis allows to determine whether two references are aliases, i.e., whether they are pointing to the same object. If two references are known to be aliases, this means that a single lock will be sufficient to protect accesses via both references. Also, if two references are known not to be aliases, modifications via one reference cannot affect the other reference. Although not mandatory, information about aliasing makes our technique more accurate because it furnishes additional hypotheses. Various tools performing alias analysis exist [37, 14].

The second analysis is a read/write one. It consists in looking where methods perform read and/or write actions. As we have seen before, JML already provides annotations for that purpose, and techniques exist to verify these annotations [13, 64, 34, 17, 63]. Typically, these techniques require an alias analysis to control reads/writes through aliases. The immutability type system presented in Section 3.2 below also includes a write-effect analysis that builds on top of an ownership type system (see also [35]). This analysis is also useful to check well-formedness of contracts i.e., comparing whether the set of locations read and written by a method contains the set of locations read by its contract.

Figure 3.3[2] (adapted from [28]) illustrates how the different analyses are used and shows why previous notions of atomicity are not satisfactory when used in a framework containing annotations. To quote Flanagan *et al.* [28], the `lookup` method "is atomic because it behaves correctly when it is concurrently invoked by multiple threads". However if we encode this "good behaviour" with a contract and try to apply contract-atomicity, we see that this is not as straightforward as they claim. Indeed, `lookup` behaves correctly in the sense that the parameter given is correctly added to the `cache`, yet our analysis detects (see next paragraph) that the value associated to the key can be computed and stored multiple times (even if the `cache` should be an optimisation!). In our encoding, the fact that a value can be computed multiple times for a similar key is detected by throwing an exception `KeyAlreadyThere`[3]. In the paragraph below,

[1] A method (of class C) is said to be a non-helper one if invariants and constraints of C must be added to its pre- and post-condition.

[2] Note that we cannot use the JML `monitors_for` keyword to specify how the `cache` is protected because this keyword protects a field of a class, whereas the policy in this example is to protect the object pointed to by the field `cache`.

[3] As mentioned in paragraph 2.3.2, our formalisation of the JMM does not consider exceptions yet, but it does not affect our explanations here.

```
Cache cache; // modifications of the cache are protected by "lock"
Lock lock;

//@ accessible cache, k;
Object get(String k);
//@ accessible cache, k, val;
void put(String k, Object val) throws KeyAlreadyThere;

Object compute(String k);

//@ requires \lockset.has(lock);
Object lookup(String k) throws KeyAlreadyThere {
    Object r = get(k);
    lock.release();

    if(r != null)
        return r;

    r = compute(k);

    lock.acquire();
    put(k, r);
    lock.release();

    return r;
}
```

Figure 3.3: Cache implementation

we explain how a contract-atomicity analysis can be used to show that `lookup` has a limited good behaviour i.e., it may throw a `KeyAlreadyThere` exception because its implementation does not prevent the value associated to a key to be computed and put into the cache more than once. Alternatively stated, `lookup` is contract-atomic w.r.t. a contract ensuring that the key is correctly put in the cache, however it is not contract-atomic w.r.t. a contract ensuring that no exception is raised.

By inspecting the method declarations we know that the only method that may throw an exception is `put`. It will throw this exception when it is called twice in a row with the same key. Thus, whether `lookup` throws an exception solely depends on the call to `put`. As indicated in comments, no method calls or field assignments can be performed on the `cache` without holding `lock`. As `lock` is held at the entry point of `lookup` (as specified by `requires \lockset.has(lock)`), we know that the cache cannot be modified until the first `lock.release()`. However, after that point, concurrent calls to `lookup` with the same key `k` can be triggered by concurrent threads. Furthermore, since `put` reads from the cache (as indicated by the annotation `accessible cache`), its behaviour depends on the state of the cache. Thus, the cache can be written by other threads between the first lock release and the call to `put`. Thus if `lookup` is called concurrently with the same key and proceeds completely, `put` will be called twice with the same key and the second thread making the call will raise an exception. That is why it is interesting to make the good behaviour of a method explicit in its contract. With the notion of contract-atomicity, it is clear what properties methods enjoy, whereas previous approaches on atomicity were not precise on this topic.

Now that we have seen how contracts help to understand acceptable behaviours of methods, we sketch how, given a method m and its contract c, we check that m is contract-atomic *w.r.t.* c. For that purpose, consider the example in Figure 3.4. Notice that this method is not considered atomic, using the classical no-

```
Point o1; // accesses to o1 are protected by "lock"
Point o2; // o2 is not protected
Lock lock;

//@ accessible o.y;
/*@ pure @*/ boolean P(Point o);

//@ accessible p2.x;
//@ assignable p1.x;
void f(Point p1, Point p2);

//@ requires \lockset.has(lock);
//@ ensures P(\result);
Point g(){
    Point r = new Point();

    f(r, o1);

    f(r, o2);

    return r;
}
```

Figure 3.4: Example to illustrate checking of contract-atomicity

tions of atomicity, because the statement `f(r, o2)` reads to `o2` (as indicated by the annotation `accessible p2.x` where `p2` is replaced by `o2` when substituting formal parameters) which can be concurrently written by other threads (because no confinement or locking discipline provides guarantees about concurrent accesses to `o2`).

The code has already been annotated with `accessible` and `assignable` clauses, to indicate where methods read and write. These annotations can be supplied by the programmer and checked by an analysis or they can be generated by the analysis itself. In addition, we consider that an alias analysis has determined that `o1` and `o2` point to different objects (i.e., they are not aliases). Also, fields `x` and `y` of `Point` objects cannot be aliases because they have type `int` (see definition of class `Point` in page 40). By performing a simple read/write analysis, we can deduce that the contract of `g` relies on the object pointed to by `r`, because the post-condition of `g` (`ensures P(\result)`) uses the method's result, which is the variable `r`. As indicated in comments, accesses to `o1` are protected by `lock`, i.e., no threads may call a method, read or change a field of `o1` without holding `lock`. Below, we distinguish between values that can be modified by the environment (called *unstable*, marked grey), and on which the current thread cannot rely, and values that cannot be modified by the environment (called *stable*, marked white).

When method `g` is entered, `lock` is held (as specified by the `requires \lockset.has(lock)`), thus the fields of `o1` cannot be changed by the environment (since `o1` is protected from unsynchronised accesses by `lock`). When the first call to `f` is made, `r` is also protected from the environment because it has not yet been shared. Thus, at that point, `r` and `o1` are stable, as depicted in graph 1 in Figure 3.5 (where only relevant information is displayed). The second call to `f` changes this situation, since it involves `o2` which is unstable. Since `f` writes to `r.x`, this also becomes unstable (graph 2). However, the method is contract-atomic because the post-condition only involves `P(r)` (as indicated by the dashed arrow), whose return value depends only on `r.y` (see graph 3) and `r.y` is stable (because it is marked by a white circle, i.e., its value has not been influenced by the environment). Note that we did not need any information about `f`, except the locations where it reads and writes.

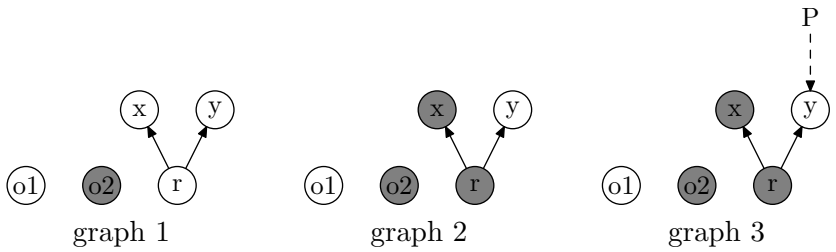


Figure 3.5: Graphs modelling partitioning of the state in unstable and stable values

Thus we can verify whether g respects its contract, using sequential program verification techniques, and because g is contract-atomic, we know that it also respects its contract in a concurrent setting.

3.1.3 Contract-independence

A notion that is closely related to atomicity is that of *independence*, introduced by Rodríguez et al. [62] for the case where all instructions in a method are both-movers (cf. [51]). Independence is useful in that it can help to verify atomicity in a modular way. If a method is independent, then all its statements access a part of the heap which cannot be concurrently accessed by other threads. This means in particular that the method does not suffer from interference, thus its behaviour is predictable, from whatever context it is called.

As with atomicity, the existing criterion to detect that a method is independent relies solely on the implementation and does not take the contract into account. However, in our framework, contracts can be used to give an alternative definition of independence:

Definition 3.1.5 (Contract-independence)**.** *A method m is* contract-independent *if for all well-formed contracts c* w.r.t. *m, if c is respected in a sequential setting, c is also respected in a concurrent setting.*

This definition allows us to re-use the existing notion of independence and we have the property that contract-independence is equivalent to independence for well-formed contracts. In addition, one can view contract-atomicity as the existential part of this definition: a method is contract-atomic *w.r.t.* a *particular* contract whereas contract-independence is concerned with *all* well-formed contracts. The motivation to have contract-independence is that it can be verified easily. Indeed, as we have seen above, contract-atomicity involves complex reasoning about read/write, aliases, locking and confinement. Contract-independence is much easier to verify and we expect it to be a useful notion to specify methods which have a strong locking or confinement policy. In fact, the both-movers criterion, initiated by Lipton's theory of reduction [51] and re-used in many frameworks ([31, 62]), is a simple way to show contract-independence. This allows us to re-use existing techniques and tools [28] to prove contract-independence. In a nutshell, this criterion consists in verifying that all possible executions of a method only involve objects that cannot be accessed at all by other threads. It is further work to re-instantiate the existing techniques to check this criterion in our framework but we expect the keywords defined in Chapter 4 to be sufficient for that purpose.

In addition, once the implementation of a method has been proven contract-independent, the programmer can change its contract without the need to check contract-independence again, solely well-formedness must be checked again. Notice that this is not true for contract-atomicity: if a method m has been checked contract-atomic *w.r.t.* c and c is changed to c' it does not entail that m is contract-atomic *w.r.t.* c'.

3.2 Immutability

In order to facilitate practical modular verification of multithreaded programs, it is important to take advantage of program properties that are not usually expressed in terms of a program logic. We have argued in previous sections that various forms of atomicity are particularly useful. We have also argued

that static analyses and type systems for controlling read and write-effects are useful for statically enforcing atomicity. An extreme form of write-effect control is the identification of immutable regions, i.e., parts of the heap that are read-only after their initialisation. Methods that do not write at all and only read from immutable regions are Lipton-reducible (even both-movers) and therefore atomic.

In an object-oriented language, it is natural to associate each object with a region of the heap—its *object state*. Objects are called *immutable* whenever they do not permit visible mutations of their object state. Many Java style guides and popular Java programming guides like Bloch's book *Effective Java* [8] recommend favouring immutable objects. One of the main reasons for this recommendation is that *immutable objects are inherently thread-safe* ([8], pg. 65).

One of the accomplishments of the first phase of this work task is the design of a static type and effect system that captures conditions for programming immutable objects [35]. We have formalised this system for a small model language that extends Featherweight Java [39] with a mutable heap. We have shown that our type system is sound in the following sense: *in well-typed programs the state of objects of immutable type does not visibly mutate.* Our type system even ensures soundness in an *open world* where clients of immutable objects are expected to follow Java's typing rules but not the rules of our special immutability type system. This open world model is useful for dealing with untrusted and legacy code. We are now planning to scale our type system to full Java and implement a prototype type checker. As a result, we will be able to exploit checked immutability specifications towards our ultimate goal: modular verification of multithreaded programs.

In the remainder of this section, we summarise important features of our immutability type system. For details we refer to [35].

3.2.1 Features of the Immutability Type System

The central keyword of our immutability system is a class annotation `immutable`, which specifies that all instances of the annotated class are immutable objects. A simple example of an immutable class is a wrapper class for integer values, similar to Java's immutable `Integer` class:

```
/*@immutable@*/ class MyInteger {
  final int value;
  MyInteger(int value) { this.value = value; }
  int get() { return value; }
}
```

Deep object states. In simple cases, an object's state merely consists of its fields, like for `MyInteger` objects. In such cases, it is enough to require that all fields are `final` in order to ensure object immutability. In more complex cases, however, an object's state naturally includes the states of objects that its fields refer to. If, for instance, object o represents a graph that is implemented as an array of adjacency lists, then o's state naturally includes not only o's fields, but also all array fields and the states of the adjacency lists. The state of each adjacency list in turn contains the state of each of its nodes. Thus, in general, object states are not disjoint but form a hierarchy ordered by inclusion.

In our Java extension, programmers can use types to specify the depths of object states. *Object types* are of the form C`<`x`>`, where C is a class name and x is an identifier that refers to the owner of `this`. Ownerless objects have types of the form C`<world>`, where `world` is a special constant. *Field types* have one of the forms C`<world>`, C`<this>` or C`<myowner>`[4]. The variable `myowner` is a special variable (like `this`) and refers to the owner of `this`. The `myowner` variable gets instantiated when a new object is created: `new` C`<`o`>`() creates a new object of type C`<`o`>`. The state of an object o consists of its fields and, recursively, of the states of all objects that are owned by o and reachable from o's fields through reference chains. Ownership typing rules ensure that objects can only have static references to their children and peers in the ownership

[4] C`<this>` and C`<myowner>` correspond to `rep` C and `peer` C in the the Universe type system [55].

tree. This is exactly like in Müller's Universe type system [55]. For instance, in the following example the state of list `l` consists of `l`'s fields and the fields of `l`'s nodes:

```
class List {
  Node/*@<this>@*/ head;
  int length;
  void cons(Object/*@<world>@*/ o) {
    head = new Node/*@<this>@*/(o,head);
    length++;
  }
}
class Node {
  Object/*@<world>@*/ value;
  Node/*@<myowner>@*/ next;
  Node(Object/*@<world>@*/ value, Node/*@<myowner>@*/ next) {
    this.value = value; this.next = next;
  }
}
Object/*@<world>@*/ o = ...;
List/*@<world>@*/ l = new List/*@<world>@*/();
l.cons(o); l.cons(o); l.cons(o);  // now l has three nodes
```

The following class `Immutable` is a simple example of an immutable class whose instances have deep object states:

```
/*@immutable@*/ class Immutable {
  final Mutable/*@<this>@*/ mtbl;      // an encapsluated mutable subcomponent
  Immutable(Mutable/*@<world>@*/ mtbl) {
    this.mtbl = new Mutable/*@<this>@*/(mtbl.get()); }
  int get() { return mtbl.get(); }
}
class Mutable {
  int value;
  Mutable(int value) { set(value); }
  int get() { return value; }
  void set(int value) { this.value = value; }  // a mutator
}
```

The type annotation `Mutable<this>` on the field `Immutable.mtbl` specifies that the state of an `Immutable` includes the state of the object that its `mtbl`-field refers to. Note that the constructor `Immutable(mtbl)` makes a defensive copy of its parameter `mtbl` to prevent representation exposure. This is enforced by the ownership type system. Technically, this is achieved because the constructor parameter's type `Mutable<world>` is not a subtype of `Mutable<this>` and, thus, a direct assignment to the field `this.mtbl` is disallowed.

Read-only methods. Obviously, methods of an immutable object should not modify its object state. One could try to ensure this by requiring that methods of immutable objects are side-effect free. However, ensuring side-effect freeness is not so simple, because even side-effect free methods must be allowed to call constructors that write to the heap. Limiting constructor writes for side-effect freeness in a practical and safe way requires alias control [63]. Therefore, instead of requiring side-effect freeness, we use a weaker restriction that is simpler to enforce on top of the ownership infrastructure and ensures that methods of immutable objects do not write to the receiver's state.

> An expression is *read-only*, if (1) it contains no field assignments; (2) all its method calls have the form $e.m(\bar{e})$, where either (a) m is read-only, or (b) e has type C`<world>`; and (3) all its `new`-calls take the form `new` C`<world>`$(\bar{e})$.

Read-only methods are guaranteed not to write to the state of immutable receivers. Our type system requires methods of `immutable` classes to be read-only. Our read-only restriction allows important side-effecting methods. For instance, the method `getChars(int srcBegin,int srcEnd,char[] dst,int dstBegin)` from Java's immutable `String` class writes to the array `dst` (owned by `world`). It is an example of a read-only method that is not side-effect free.

Write-local, anonymous constructors. A constructor of an immutable object typically will have side-effects to initialise the object state. We have to restrict constructors of immutable objects for two reasons: (i) we have to prevent them from modifying other objects of the same class, (ii) we have to prevent them from leaking the partially constructed `this`.

Restriction (i) is needed because visibility modifiers in Java constrain per-class, not per-object, visibility. For this reason, unrestricted constructors of immutable objects could modify other immutable objects of the same class. For example:

```
/*@immutable@*/ class Wrong {
  Mutable/*@<this>@*/ mtbl;
  int get() { return mtbl.get(); }
  Wrong(Wrong/*@<world>@*/ o) {
    this.mtbl = new Mutable/*@<this>@*/(o.get());
    o.mtbl.set(23);  // unwanted side-effect on other object!
  }
}
```

To prevent such immutability violations, we require constructors of `immutable` objects to be write-local in the following sense:

> An expression is *write-local*, if (1) all its field assignments have the form $e.f$=e' where either $e =$ `this` or e has a type C`<this>` and (2) all its method calls have the form $e.m(\bar{e})$ where either (a) m is read-only or (b) m is write-local and $e =$ `this` or (c) m is write-local and e has type C`<this>` or (d) e has type C`<world>`.

Restriction (ii) is needed to prevent immutable objects from making themselves visible during their construction phase when they are still mutating. We use Vitek et al.'s notion of anonymity to prevent constructors of immutable objects from leaking `this` [67, 69]:

> An expression is *anonymous*, if it (1) is not `this`, (2) does not pass `this` as a method argument unless the receiver is `this`, (3) does not assign `this` to fields, and (4) all its method calls have the form $e.m(\bar{e})$ where either e or m is anonymous.

Our type system requires constructors of `immutable` classes to be both write-local and anonymous.

Owner-polymorphic methods for safe dynamic aliasing during object construction. Methods can have owner parameters. Like type parameters of Java 5's generic methods, these are ignored at run-time and are used for type-checking only. Declarations of owner-polymorphic methods are of the form $\texttt{<}\bar{y}\texttt{>}\ ty\ m(\bar{ty}\ \bar{x})\{e\}$. The scope of the owner parameters $\bar{y}$ includes the types $T, \bar{ty}$ and the method body e. The type system restricts occurrences of owner parameters within e to inside angle brackets $\texttt{<}\cdot\texttt{>}$. Calls of owner-polymorphic methods must explicitly instantiate the owner parameters: in the method call expression $e.m\texttt{<}\bar{v}\texttt{>}(\bar{e})$, the values $\bar{v}$ instantiate the owner parameters $\bar{y}$ from m's declaration.

Owner-polymorphic methods permit *dynamic aliasing of representation objects*. Consider, for instance, the following method declaration:

```
<x,y> void copy(C<x> from, C<y> to)
```

A client may invoke the `copy` method with one or both of `x` and `y` instantiated to `this`, for instance, `copy<world,this>(o,mine)` where `mine` refers to an internal representation object owned by the client. When a `copy`-client makes this call, he passes a dynamic alias to his representation object `mine` to the `copy`-method. Dynamic aliasing of representation objects is often dangerous, but can sometimes be useful. For immutability, dynamic aliasing is useful during the object construction phase, but dangerous thereafter. For instance, the constructor `String(char[] a)` of Java's immutable `String` class passes an alias to the string's internal character array to a global `arraycopy()` method, which does the job of defensively copying `a`'s elements to the string's representation array. Our type system uses owner-polymorphic methods to permit dynamic aliasing during the construction phase of immutable objects, but prohibits it thereafter. The latter is achieved by prohibiting read-only expressions to instantiate a method's owner parameters by anything but `world`.

For `String` to be immutable in an open world, it is important that the `arraycopy()` method does not create a static alias to the representation array that is handed to it from the constructor `String(char[] a)`. Fortunately, owner-polymorphic methods prohibit the creation of dangerous static aliases! This is enforced merely by the type signature. Consider again the `copy()` method: From the owner-polymorphic type we can infer that an implementation of `copy` does not introduce an alias to the `to`-object from inside the transitive reach of the `from`-object. This is so, because all fields in `from`'s reach have types of the form D`<x>` or D`<from>` or D`<world>` or D`<`o`>` where o is in `from`'s reach. None of these are supertypes of C`<y>`, even if D is a supertype of C. Therefore, `copy`'s polymorphic type forbids assigning the `to`-object to fields inside `from`'s reach.

Read-only objects for safe sharing of mutable representation objects. Our type system distinguishes between read-only objects and read-write objects. Although read-only objects may have mutator methods, clients of read-only objects are not allowed to call these and, consequently, the state of read-only objects does not mutate after initialisation. Thus, read-only objects are very similar to immutable objects. The important difference is that read-only objects cannot safely be exposed to untrusted clients, because we cannot expect that untrusted clients follow our special read-only rules. Immutable objects, on the other hand, can safely be exposed to arbitrary Java clients, because even untrusted Java clients have no means to mutate their state.

Our motivation for adding read-only objects to our system is that we want to support sharing mutable representation objects among different immutable objects. Our type system permits this kind of sharing as long as the shared objects are read-only.

Technically, we realise read-only objects by enriching object types with an *access right parameter*. Enriched object types are of the form C`<`ar, o`>` where ar ranges over the *access right constants* `rdwr` ("read-write") and `rd` ("read"). Read-only objects owned by o have types of the form C`<rd,`o`>`. The access right parameter may be omitted and is initially set to `rdwr`. In addition to the access right constants `rdwr` and `rd`, there is a *special access right variable* `myaccess` that refers to the access rights of `this`.

In order to permit sharing of mutable representation objects between immutable objects, we postulate that types of the forms C`<rd,`o`>` and C`<rd,`p`>` are equivalent in case o and p have `immutable` types. In order to prevent that our type system restricts untrusted Java clients, we postulate that types of the forms C`<`ar`,world>` and C`<`ar'`,world>` are equivalent for arbitrary access rights ar, ar'. Our type soundness proof shows that these type equivalences do not break soundness of our type system.

The following simple example illustrates sharing of mutable representation objects. In the second constructor, the assignment of `o.mtbl` to `this.mtbl` is allowed because the types `SharedRepObject<rd,o>` and `SharedRepObject<rd,this>` are equivalent.

```
/*@immutable@*/ class SharedRepObject {
  Mutable/*<rd,this>*/ mtbl;
  int get() { return mtbl.get(); }
```

```
  SharedRepObject (int i) {
    this.mtbl = new Mutable/*<rd,this>*/(i);
  }
  SharedRepObject (SharedRepObject/*<world>*/ o) {
    this.mtbl = o.mtbl;  // o and this now share a representation object
  }
}
```

3.3 Thread Ownership

Another condition that can help to support thread-modular verification is the notion of thread ownership. If an object is known to be local to a thread (i.e., there is only one thread that can access an object), we do not need to synchronise access to it. The literature contains various proposals for ownership type systems, but often these do not consider thread ownership. This section proposes an advanced thread ownership system that precisely tracks how many threads may access an object. In contrast to standard ownership systems, for thread ownership systems the transfer of ownership is a crucial element; one cannot expect that objects are owned by the same thread throughout the execution of the application.

3.3.1 Specifying Locality with Capacities

Thread locality is concerned with the ownership of objects by threads. In multithreaded programs, some objects may be shared between different threads, while others are local to a particular thread. It is important to know this information, because it can help to simplify verification. For example, if we know that all objects involved in a method call are local to the current thread, one can assume a sequential context when verifying the correctness of this method. In previous work in this area [27, 9] objects are either local or shared and once an object is shared it cannot become local again. Yet, many typical concurrent programming patterns (see [47]) do not respect this restriction. For example, a thread t_1 may share an object o with a newly created thread t_2 and retrieve o locally once t_2 has died.

Therefore, this section proposes a means to annotate a program, so that we can express and verify properties related to thread locality. Various results on ownership have emerged in the last years (see e.g., [18, 23]), however these focus on *object* ownership, whereas we address *thread* ownership. An elegant framework for thread ownership has been pioneered by Boyland [10], and we think this can be extended for multithreaded programs.

We use an annotation `shared<x,y>` to indicate whether an object is intended to be shared or not; `x` indicates the maximum number of threads that may execute simultaneously within the object, while `y` indicates the maximum number of threads that may execute simultaneously, without the permission to write. The annotation can be written as a class modifier, a field modifier or as a parameter annotation.

If the annotation `shared<x,y>` is written as a class modifier, it means that `x` different threads may execute simultaneously (i.e., call methods and access or update fields) for every instance of this class, while at most `y` different threads can execute simultaneously in read-only mode (i.e., calling pure methods and reading fields). For example, a class implementing a shared container would be annotated `shared<2,0>` to mean that it may be accessed simultaneously by two writing threads. This annotation applies to every object that is an instance of the class. We say that it specifies the *total capacity* of an object. This is a static notion, which does not change during execution. Figure 3.7 on page 51 specifies that the total capacity of class `Task` is `shared<1,1>`, i.e., there is at most one thread writing and at most one thread reading the task at any point in time.

Each reference to such an object also has a capacity. The capacity of a reference can change during program execution, by assigning different objects to it. However, at any point the capacity of the reference has to be less or equal than the total capacity of the object it is pointing to. Therefore, we call the capacity of a reference a *partial capacity*. Moreover, the sum of the partial capacities of all references to a single

```
class Line3D {

    ReentrantLock l;

    /*@ shared(1,0)<1,1> @*/ Point3D p;
    /*@ shared(1,0)<1,1> @*/ Point3D q;

    //@ requires \lockset.has(l);
    //@ ensures (p.x == \old(p.x) + x) && (q.x == \old(q.x) + x);
    public void xShift(int x) {
        p.x += x;
        q.x += x;
    }
}
```

Figure 3.6: Class `Line3D`

object can never exceed the total capacity of the object. Comparison, addition and subtraction of capacities is defined formally in the next subsection.

To distinguish between the partial and total capacities of a reference, both fields and parameters are annotated with two capacities: partial capacities are enclosed between round brackets "`()`" whereas total capacities are enclosed between angle brackets "`<>`". The field declarations in Figure 3.6 show the use of capacities as field modifiers.

The annotation `shared(1,0)<1,1>` indicates the following:

- objects pointed to by references `p` and `q` must have a static type greater or equal than `shared<1,1> Point3D`, because the total required capacity is at least `shared<1,1>`; and
- because of `shared(1,0)` at most one thread at the time can call a method or access a field on the instance of class `Point3D` pointed to via `p` and `q`.

This thread ownership information can be exploited to establish the correctness of method `xShift`. Without any synchronisation, the post-condition of `xShift` can be violated in several ways, e.g., between the execution of the last instruction of `xShift` and the evaluation of its post-condition, another thread can modify the field `x` of the object pointed to by `p` or `q`. However, because of the annotation `shared(1,0)<1,1>`, no other thread can simultaneously write a field of the objects pointed to by `p` and `q` when a thread is executing `xShift` (however, they are allowed to be read simultaneously). Thus, within `xShift` the value of `p.x` is stable, and the post-condition of `xShift` can be proven correct.

Thus to recapitulate, when `shared(x,y)<w,z>` annotates a field declaration, this means that `x` (resp. `y`) different threads may execute (resp. execute in read-only mode) concurrently in the object pointed to by that reference. In addition, the total capacity of the object pointed to by that reference must be less or equal than `shared<w,z>`.

Finally, a similar syntax is used to write capacities in method declarations. A parameter `p` annotated with `shared(x,y)<w,z>` indicates that the object passed as a parameter to this method must have a total capacity less or equal than `shared<w,z>` and that the reference used when passing the parameter must have a partial capacity greater or equal than `shared(x,y)`.

Figure 3.7, which is an implementation of a worked-thread-model (inspired by Doug Lea [47]), illustrates the use of this notation. The main thread creates a `Worker` thread to perform a task and an `Observer` thread to observe what the worker is doing.

As mentioned above, the annotation `shared<1,1>` in class `Task` indicates that at most one thread with write permission and at most one thread with read permission may execute simultaneously within any

```
class Main{

    public static void main(String[] args){

        Task t = new Task();

        Thread t1 = new Worker(t);
        Thread t2 = new Observer(t);

        t1.start();
        t2.start();
    }
}

class /*@ shared<1,1> @*/ Task {

    void doTheJob(){}

    /*@ pure @*/ void observe(){}
}

class Worker extends Thread {

    /*@ shared(1,0)<1,1> @*/ Task t;

    public Worker(/*@ shared(1,0)<1,1> @*/Task t){
        this.t = t;
    }

    public void run(){
        t.doTheJob();
    }
}

class Observer extends Thread {

    /*@ shared(0,1)<1,1> @*/ Task t;

    public Observer(/*@ shared(0,1)<1,1> @*/Task t){
        this.t = t;
    }

    public /*@ pure @*/ void run(){
        t.observe();
    }
}
```

Figure 3.7: Worker threads example

instance of `Task`. The declaration `shared(1,0)<1,1>` on field `t` in class `Worker` indicates that the worker thread has the right to execute in the task `t`. Further, the annotation `shared(1,0)<1,1>` in the constructor of class `Worker` indicates that the object passed when constructing a worker thread must have a total capacity less or equal than `shared<1,1>`, and that the actual parameter reference should have a partial capacity greater or equal than `shared(1,0)`.

Intuitively, when the main thread constructs a `Worker` thread, it gives the capacity `shared(1,0)` to

the newly created thread. When the `Observer` thread is created, the main thread gives it the capacity `shared(0,1)`. Thanks to this technique, verification of code in the `Worker` thread can assume that no other threads will write to the task simultaneously (because the annotation `shared(1,0)<1,1>` on the task `t` in the `Worker` classes indicates that remaining capacities are such that the environment has only the right to read from the task). More generally, having information about both partial and total capacities permits to know what the environment has the right to do. That is why, even if the total capacity is an information about the whole program, it allows to perform a modular reasoning. The difference between partial and total capacities permits to know what the rest of the program is allowed to do, in a way that is abstract enough to keep the analysis modular. In particular, we do not know how the whole program is organised. For example, if the difference between the total and the partial capacity of an object is $(3, 2)$, we do not need to know whether this is divided in partial capacities $(1, 0)$ and $(2, 2)$ or $(3, 1)$ and $(0, 1)$ to be able to exploit this information.

In read-world programs, a wide variety of synchronisation patterns exist. An important one among them is protection by external locks [27, 9]. An object is externally protected if all threads agree to acquire a list of locks before performing any action (method call, field read/write) on that object. Using this pattern is possible in our framework by attaching policies (enclosed in braces) to capacities. The notation `shared(x,y)<w,z>{l1, ..., ln}` (where for all `i` in [1, n], `li` denotes a lock) indicates that all threads must respect the policy to synchronise on lock `l1` and ... and `ln` before calling a method or read or write a field of the object pointed to by that reference. Policies can be specified wherever capacities can appear. When it is next to a class declaration, `shared<x,y>{l1, ..., ln}` indicates that, when an instance of this class is created, `l1` to `ln` must designate non-null locks at runtime, and that accesses to instances of this class can only be performed when `l1` to `ln` are held. When it is next to a field or a parameter, the annotation `shared(x,y)<w,z>{l1, ..., ln}` denotes the additional constraint that `l1` to `ln` must designate non-null locks, and any accesses to the reference can only be performed when all locks in the list are held.

For the purpose of modular verification, the system of partial/total capacities and policies can be used to make verification simpler. In fact, when a method m of class C is verified, for all fields (of C) or parameters (of m) o such that the difference between the total and the partial capacity of o is equal to $(0, y)$ (where y can be any value), the verification can safely assume that all fields of o are stable during the analysis. In particular, if a value of o is cached or written during the execution of m, this value is not invalidated by the environment (see explanations about the verification of method `xShift` in Section 3.6). Policies allow the same kind of reasoning within program fragments respecting them: when a method m is verified, for all field or parameters o of the class of m whose policies is respected (i.e., all objects and locks specified in o's policy are held by the thread being observed), the verification can safely assume that all fields of o are stable during the program fragment considered. Consider, for example, the following piece of code.

```
//@ requires c != null && \lockset.has(c);
//@ ensures s == c.size;
public MyVector(/*@ shared(x,y)<w,z>{c} @*/ MyVector c){
    s     = c.size;
    data  = new Object[elementCount];
    c.toArray(data);
}
```

The analysis can assume that `s == c.size` is true after the first statement because the environment cannot perform method calls on `c`. This is ensured by the policy `{c}` and the fact that `c` is held during the execution of this method (as specified by the pre-condition `\lockset.has(c)`). Without the policy, the possibility of concurrent calls on `c` would make it impossible to assume that `s == c.size` is true after the execution of the first statement.

Note that the system of capacities does not prevent data races. In the example from Figure 3.6, the annotation `shared(1,0)<1,1>` on fields `p` and `q` makes sure that other threads can only access `p` or `q` in read-only mode. Thus, no synchronisation is needed to make sure that the values of variables are not changed.

However, if we take the point of view of these "read-only" threads, they have capacity `shared(0,1)<1,1>` on the objects pointed to by `p` and `q`, and so, they are aware that concurrent writes are possible. In order to avoid data races, synchronisation should be enforced by the use of policies.

Finally we introduce some syntactical shortcuts to make it easier for the programmer to annotate the code. These shortcuts can be used to annotate fields and parameters and they all desugar into `shared`. In practice these annotations appear often and permit to capture more easily the intended behaviour of a program.

shortcut		desugaring
`local`	$\triangleq$	`shared(1,0)<1,0>`
`unvarying<y>`	$\triangleq$	`shared(0,1)<0,y>`
`readonly<x,y>`	$\triangleq$	`shared(0,1)<x,y>`
`unusable<x,y>`	$\triangleq$	`shared(0,0)<x,y>`
`shielded<y>`	$\triangleq$	`shared(1,0)<1,y>`

Informally, `unvarying` means that no thread, not even the current one, can modify the object (this notion is similar to immutability as presented in Section 3.2), `readonly` means that the current thread cannot modify the object but the environment has the right to do so, `unusable` indicates that the current thread cannot read or write to the object annotated (this is useful to indicate that a concurrent container may not modify objects stored) and `shielded` indicates that the current thread is the only one with the right to write. Finally, `local` can also be used as a class annotation, in this case it desugars to `shared<1,0>`, meaning that only one thread at a time can execute within any instance of the class annotated.

3.3.2 Ordering and Updates of Partial Capacities

When a program is executed, capacities are updated according to the code processed. In the following example, after the execution of `t2.start()`, the main thread cannot execute a method or write a field of the task, otherwise it would violate the constraint imposed by the annotation `shared(1,1)` in class `Task`. The following code fragment shows (in comments) how the capacity of the reference `t` is updated during execution.

```
class Main{

    public static void main(String[] args){

        Task t = new Task();
        // t is shared(1,1)<1,1>

        Thread t1 = new Worker(t);
        Thread t2 = new Observer(t);

        t1.start();
        // t is shielded<1> i.e. shared(1,0)<1,1>
        t2.start();
        // t is unusable<1,1> i.e. shared(0,0)<1,1>
    }
}
```

After the call to `t2.start()`, `t` has partial capacity `shared(0,0)`, i.e., the main thread cannot execute any operation on `t`. This shows how capacities ensure the correctness of the program by avoiding that there are 3 threads simultaneously executing within the task.

More precisely, capacities are updated by using their ordering. A capacity c_1 = `shared(x1,y1)` is greater or equal than a capacity c_2 = `shared(x2,y2)` (denoted $c_1 \geq c_2$) if `x1` > `x2` or if `x1` = `x2` and `y1` ≥ `y2`. The subtraction of capacities is defined as follows: $c_1 - c_2$ = `shared(x1 - x2,y1 - y2)`, provided $c_1 \geq c_2$. Similarly, capacities can be added: $c_1 + c_2$ = `shared(x1 + x2, y1 + y2)`. Capacities are updated whenever a communication occurs between different threads. Communications can be thread start-up/death, method calls or assignments on objects that are or will be shared. For example, if `o` is an object shared among different threads, an assignment to a field of `o` is a communication, as well as assigning an object `o2` to an object `o1` and sharing `o1` among different threads. A conservative approach is to consider that any assignment, method call or object creation is a communication. In order to understand how capacities are updated when communications take place, consider the following generic code fragment:

```
static public void main(String[] args){

    Object o1;
    // o1 is shared(x1, y1)<w1, z1>

    t = new MyThread(o1);
    t.start();
    // o1 is shared(x1 - x2, y1 - y2)<w1, z1>

}

class MyThread extends Thread {

    MyThread(/*@ shared(x2, y2)<w2, z2> @*/ Object o2){
        ...
    }

}
```

When the main thread executes `t.start()`, if `shared(w1,z1)` > `shared(w2,z2)` the program is incorrect because of a type mismatch between the total capacity of object `o1` and the total capacity specified in `MyThread`'s constructor. If `shared(x1,y1)` < `shared(x2,y2)` the program is incorrect, because the new thread requires a larger capacity than the main thread provides. In all other cases the program is correct *w.r.t.* the capacity and the partial capacity of reference `o1` is updated to `shared(w2 - x2,z2 - y2)` in the main thread and the partial capacity of the reference `o2` is set to `shared(x2,y2)`. Thus the decrease of the capacity of the task in the main thread models that the main thread shares its rights on `o1` and the capacity transferred to the new thread models how threads acquire rights. Similarly when a thread t_1 waits for another thread t_2 to die (*via* `join()`), capacities of objects within t_2 can (under some additional conditions) be reclaimed by t_1. In this case, the fact that t_1 acquires some capacities is modelled by summing capacities.

The ordering of capacities is also useful to deal with inheritance in a natural way. If class B inherits from class A whose class annotation is c_A, B's class annotation must be less or equal than c_A. This is to ensure that, given an object whose static type is A, the total capacity visible in the program is always a sound approximation of the "real" total capacity of the object at run-time. If its dynamic type is A, no inheritance is involved and its total capacity is c_A (thus, equal to the total capacity visible). If its dynamic type is B, its total capacity is c_B which is less or equal than c_A and thus is a sound approximation of the total capacity visible in the program (because $c_B \leq c_A$).

We are currently developing a type-system to statically verify the correctness of annotations about locality. This type-system relies heavily on alias analysis because duplicating a reference to an object (e.g., creating an alias) should not break the correctness of annotations about locality. In order to overcome this issue, we use an approach similar to destructive reads (cf. Section 4.1 of [11]), i.e., whenever a reference is copied, capacities are potentially updated. Consider the following piece of code:

```
Box b = new Box();    // b receives capacity(1,0)<1,0>
/*@ shared(1,0)<1,0> @*/ Box c;

c = b;                  // an alias is created, b's capacity must be updated

// as c requires partial capacity shared (1,0)
// b has now capacity shared(1,0)<1,0> - shared(1,0)<1,0>
//                    = shared(0,0)<1,0>
```

When the statement `c = b` is type-checked, the capacity of `b` must be decreased otherwise there could be a violation of the annotations. For example, if `b` and `c` are later given to two different threads, these two threads would be able to execute simultaneously within the object pointed to by `b` and `c` (which would be incorrect since `Box` objects have a total capacity `<1,0>`, i.e., no more than one thread is supposed to execute concurrently in them).

A similar behaviour occurs with method calls. Imagine the current thread has complete access (i.e., the partial capacity is equal to the total one) to a variable `t`. If `t` is passed as an argument to a method call, its capacity must be updated if the method involved keeps a reference to the object pointed to by `t` (otherwise, again there would be a violation of the annotations). The notion of "keeping a reference" has already been formalised in JML with the `captures` clause. For example, a concurrent container having a `store` operation keeps a reference to the object passed as a parameter:

```
class ConcurrentContainer {

  //@ captures o;
  public void store(/*@ shared(0,1)<1,1> @*/ Object o){
    ...
    }
}
```

Calls to `store` provoke an update of the capacity of objects passed, as it is indicated in comments in the following example:

```
class MyThread extends Thread {

  ConcurrentContainer c = new ConcurrentContainer();

  void public run(){
    Task t = new Task();    // t receives capacity shared(1,1)<1,1>

    c.store(t);             // t is captured in the container

    // t has capacity shared(1,1)<1,1> - shared(0,1)<1,1>
    //                = shared(1,0)<1,1>
  }
}
```

As `store` requires partial capacity `shared(0,1)` (as indicated in the annotation of its formal parameter `o`), when a thread calls `store`, it should abandon this partial permission to the concurrent container to preserve the correctness of the locality system. Eventually, we will also study how the capacity annotations can be inferred as much as possible.

Finally, we plan to investigate if a non-modular variation of `shared` could be useful. Such an annotation would denote that all objects reachable via the object annotated with `shared` would also have to follow

the sharing policy. Thus, whenever all locks are held, no other thread can change any of the objects that can be reached via the reference. We plan to investigate how often the need for such an annotation arises in practice. Advantage of such an annotation is that it gives a strong assumption for verifying the class containing the annotation, but drawback is that verification of this annotation is non-modular, and has to be redone every time a program is extended.

3.4 Exploiting Conditions for Thread-modular Verification

In this chapter, we have explained different notions that are crucial for thread-modular verification. Immutability (Section 3.2) and thread ownership (Section 3.3) permit to ensure properties of objects. Immutable objects cannot be written by any thread while objects correctly synchronised or owned by a thread are protected against unexpected accesses. Once these notions have been checked, one is sure that a variety of interferences are ruled out.

On top of these two notions dedicated to objects, we have developed contract-atomicity (Section 3.1) to reason about method contracts in a sound and modular way. It is future work to integrate all these notions together, yet clearly, contract-atomicity relies on immutability and thread ownership. Thus, our verification process will begin by immutability and thread ownership checking, followed by a contract-atomicity analysis that enables the use of sequential program verification techniques.

Chapter 4

Specification of Multithreaded Applications

In the previous chapter, we studied several conditions for thread-modular verification, that allow us to re-use program verification techniques for sequential programs. To specify the desired behaviour of a program, we take the Java Modeling Language (JML) as a starting point, as this is a widely used specification language, with extensive tool support [12, 19, 15]. However, in its current state, JML is not suited to directly specify the behaviour of a multithreaded program. In particular, as mentioned above, in a multithreaded setting pre- and post-conditions of methods can be invalidated by concurrent threads; this is the so-called problem of *interference*. Thus we need to extend the language with the possibility to protect method specifications against interference, so that modular verification is possible, based on the techniques developed in the previous chapter. Another problem is that in a multithreaded setting one cannot directly rely on class invariants or history constraints anymore, and we need to fine-grain the semantics of these notions.

This chapter proposes a concrete list of keywords to extend JML, that allow one to specify and verify the requirements for thread-modular verification. The proposal is based on what already exists in JML, an earlier proposal by Rodríguez et al. [62], and the work presented in the previous chapter. In particular, it includes keywords for contract-atomicity, immutability and thread ownership. For each keyword, we describe why we propose it, its intended meaning, its syntax, how it can be exploited and how it can be verified. It is future work to define a complete formal semantics and appropriate verification techniques. In principle, the keywords that RCC requires to check for absence of data races (see Section 2.4 and [27]) should also be part of this language extension. However, some of the keywords used here are close to JML, but with a slightly different meaning. It is future work to study the possible interactions between the different language extensions, and to propose one uniform specification language.

Further, this chapter also discusses how multithreading has an impact on the semantics of the existing language constructs in JML, in particular on the notions of invariant and constraint. We show how the semantics of the existing specification language will have to be adapted. We also discuss some specification constructs that are already present in JML, but that will become more important in a multithreaded setting.

Finally, we compare our language with existing other proposals for specification languages for multithreaded applications, and we illustrate the use of the extension of JML by means of an example.

Throughout the discussion, there is a subtle issue about what locks exactly are. Since Java 1.5, one can use the `synchronized` statement to lock on the implicit lock that is associated with any object, but one can also explicitly declare a lock (inheriting from class `Lock`), and call methods `lock` and `unlock` on it. This is different from synchronising on the implicit lock associated with this lock object. For the sake of simplicity, we require `synchronized` expressions to be used solely on objects whose type does not inherit from `Lock` (as recommended in Sun's documentation for the class `Lock`).

4.1 Specification Keywords for Thread-modular Verification

This section describes the list of keywords that we propose as an extension to JML, and can be used to specify the necessary conditions for thread-modular verification. The keywords are given in alphabetic order. The list of keywords is based on the current version of JML, the earlier proposal by Rodríguez et al. [62], and the previous chapter. We believe this list to be sufficient to specify (and verify) any interesting thread-modular property about a multithreaded program. However, this conviction is only based on our experience in writing example specifications, and we cannot exclude that, later in the MOBIUS project, we will find that we need additional expressiveness. As mentioned above, eventually, also the keywords needed to check the absence of data races with RCC should be integrated into JML.

`captures o1, ..., on` (from [48, §9.9.13])

- *Motivation* Supports the verification of thread ownership (Section 3.3).
- *Syntax* This keyword is a method specification clause, where the variables `o1`, ..., `on` are parameters of the method (including the implicit method receiver).
- *Meaning* This keyword is already part of JML, and we use the semantics as described in the JML Reference Manual [48]: a captures clauses `captures o1, ..., on` specifies that references to `o1`, ..., `on` can be retained after execution of the method has finished, for example in a field of the receiver object. The default clause is `captures \everything`, meaning that the method is allowed to capture any of the actual parameter objects or the receiver.
- *Use* The list of objects specified in a `captures` clause is used when verifying the correctness of thread ownership annotations.
- *Verification* It is future work to develop verification techniques for this annotation. However, we expect to re-use existing work on that topic [21].

`contract_atomic` (see Section 3.1)

- *Motivation* Support for modular verification: a method that is contract-atomic does not suffer from the interference problem, i.e., it can be verified sequentially, and its specification can be used for the verification of other methods.
- *Syntax* This keyword is a behaviour specifier.
- *Meaning* As described in Section 3.1, a method is contract-atomic for a certain behaviour contract c if, whenever the contract c is respected in a sequential setting, c is also respected in a concurrent one.
- *Use* If a method m is contract-atomic for a contract c, then it can be verified sequentially whether m respects c. Moreover, when verifying a method m' calling method m, the contract c can be used, i.e., if the pre-condition of c is satisfied, the post-condition can be assumed.
- *Verification* It is ongoing work to develop a verification technique to check contract-atomicity. Alias analysis and read/write analysis can be used to improve the precision of the contract-atomicity verification.
- *Alternative syntax* The keyword `contract_atomic` can also be used as method modifier. This is desugared by adding `contract_atomic` annotations to all the method's behaviour specifications.

`contract_independent` (see Section 3.1)

- *Motivation* Contract-independence provides an easily statically enforceable way to guarantee contract-atomicity.
- *Syntax* This keyword is a method modifier.
- *Meaning* All well-formed contracts c for this method are contract-atomic.
- *Use* Contract-independence implies contract-atomicity for any well-formed contract.
- *Verification* Contract-independence can be verified using the both-mover criterion from Lipton [51].

`immutable` (see Section 3.2)

- *Motivation* Access to immutable objects does not have to be protected by synchronisation.
- *Syntax* This keyword is a class modifier.
- *Meaning* An object is immutable in a given program iff its state does not visibly mutate in any run of the program. A class is immutable iff all its instances are immutable in all programs. Our type system as outlined in Section 3.2 provides a sound but incomplete verification method with respect to this semantic definition.

  Alternatively, we could define the semantics of immutability like this: a class is immutable iff our immutability type system can prove it. This definition uses our static type system as the semantics of immutability. This approach has two pragmatic advantages: Firstly, it disallows declaring classes as immutable if our static analysis cannot prove their immutability. Experience has shown that unprovable specifications cause a lot of confusion for users of extended static checking tools. Secondly, we avoid having to get too deeply into questions like "What exactly is an invisible mutation?", which is not so easy to answer formally and precisely in the context of a large programming language like Java.

  Thus, there are different possible choices for the semantics of the `immutable` keyword. In that respect, the `immutable` keyword is similar to JML's keyword `pure`.
- *Use* The contents of an immutable object cannot be affected by any thread.
- *Verification* An extension of the immutability type system from [35] can be used to verify immutability.

`\lockset` (from [48, §11.4.19])

- *Motivation* Allows to make explicit statements about the locks that are currently being held.
- *Syntax* This keyword is a JML primary expression of type `JMLObjectSet`.
- *Meaning* This keyword is already part of JML, and we use the semantics as described in the JML Reference Manual [48]: the `\lockset` expression denotes the set of locks held by the current thread. Notice that its content is implicitly updated whenever a lock is acquired/released or a `synchronized` block is entered/left.
- *Use* The lockset is used for various analyses: contract-atomicity analysis, locality checking etc.
- *Verification* The proof obligation generator has to generate appropriate proof obligations for this keyword. In fact, ESC/Java [19] already handles this keywords, with a slightly different syntax: `\lockset[this]` instead of `\lockset.has(this)`.

`monitors_for f ← l1, ..., ln` (from [48, §8.9])

- *Motivation* Protects a single reference or field.
- *Syntax* This keyword is a class specification, where `f` is an expression, denoting a field visible in the current class, and `l1,...,ln` is a list of locks. If `l1,...,ln` are different from `this`, they have to be final.
- *Meaning* The field `f` cannot be be read or written without holding all the locks `l1,...,ln`. Notice that this does *not* recursively protect the fields of the object that `f` references. In order to protect these, one needs to explicitly specify `monitors_for` clauses, use an object ownership system to show that the fields only can be accessed via the reference `f`, or use the `shared` annotation described below.
- *Use* Specifies a locking discipline for accessing fields. It can be used to prove the absence of data races. When verifying a multithreaded program, one can assume that, whenever the lockset contains `{l1, ..., ln}`, `f` cannot be changed by another thread.
- *Verification* This keyword is already handled by ESC/Java [19], with a slightly different syntax: `monitored_by` as a field declaration modifier.
- *Alternative syntax* JML and ESC/Java have slightly different keywords to express this property: a class specification `monitors_for` versus a field declaration modifier `monitored_by`. Also the RCC tool supports a similar construct: `guarded_by`. It remains to be investigated whether it is worth supporting all these keywords, or whether we can decide on a "best" syntax.

`shared<w,z>{l1, ..., ln}` (see Section 3.3)

- *Motivation* Supports thread ownership system.
- *Syntax* This keyword is a class modifier, where `w` and `z` are natural numbers and `l1, ...,ln` are names of locks that have to be instantiated when the object is created.
- *Meaning* Only `w` (resp., `z`) different threads may simultaneously access (resp., access in read-only mode) an instance of this class, while all locks in the list `l1, ..., ln` must be held when accessing the object.
- *Use* If `w` ≤ 1 or the list `l1, ...,ln` is not empty, for any field of the current object, one can safely assume that its value cannot be affected by another thread, i.e., the value of the field can only be changed by the current thread.
- *Verification* It is ongoing work to develop a verification technique for this annotation.

`shared(x,y)<w,z>{l1, ..., ln}` (see Section 3.3)

- *Motivation* Supports thread ownership system.
- *Syntax* This keyword is a variable modifier, i.e., it can specify both field and parameter declarations. The parameters `x`, `y`, `w` and `z` are natural numbers, while `l1, ...,ln` is a list of locks visible in the current class (for field declarations) or in the method (for parameter declarations).
- *Meaning* Only `x` (resp., `y`) different threads may execute (resp., execute in read-only mode) concurrently in the object pointed to by the reference. Moreover, all locks in the list `l1,..., ln` must be held before executing within the object pointed to. The fields `w` and `z` denote the total write and read-only capacity of the object: the difference with `x` and `y`, respectively, denotes the number of other threads that might be accessing the same object simultaneously.

- *Use* This annotation is closely related to the `shared` class annotation. For any field or parameter `o` of class `C`, annotated with `shared(x,y)<w,z>{l1, ..., ln}`, if the lockset contains `{l1, ..., ln}` or if `x` = `w`, then, for any field `f` of `C`, one can safely assume that `o.f` cannot be changed by any concurrent thread.
- *Verification* It is ongoing work to develop a verification technique for this annotation.

4.2 Using JML for Multithreaded Applications

As mentioned above, in a multithreaded setting one cannot rely on class invariants or history constraints anymore. In a sequential program, a class invariant is a predicate that holds in all visible program states, i.e., in all states in which a method is called or finished. Within a method body, an invariant might be temporarily broken. However, in a multithreaded program, one thread might call a method that depends on an invariant, while another thread is in the middle of a method that temporarily breaks this invariant. Thus, the method that is being called in the first thread might end up in an inconsistent state, because the invariant does not hold when the method is called. It is future work to study in detail the possible solutions to this problem. One possibility is to add a notion of *strong* invariant, i.e., an invariant that is never broken, not even temporarily. Another possibility is to specify explicitly properties on which a thread can rely when it has certain locks. This could be expressed using the following keyword.

`locking_rely(l1, ..., ln) P`

- *Syntax* This is a class specification, where `l1, ..., ln` are objects in the scope of the current class and `P` is a JML predicate.
- *Meaning* If a thread holds the locks `l1 ...,ln`, it can assume the property `P` to hold. Alternatively stated, the environment guarantees `P` when `l1,...,ln` are held.
- *Use* If the lockset contains `{l1, ..., ln}`, one can assume that `P` holds.
- *Verification* Whenever a thread wants to release any of the locks `l1, ...,ln`, it has to ensure that `P` holds. This should be an explicit verification condition for the current thread.

One can also imagine further combinations of the two, where the current thread has to ensure that the predicate `P` holds in any of its visible states. It is future work to study further how the notion of invariant can be integrated (and exploited) in a multithreaded setting.

For constraints there is a related problem. A constraint describes a relation that is supposed to hold between every pair of visible states: however in a multithreaded setting this might be visible states belonging to different threads. Thus, to verify a constraint one has to consider all possible interleavings of the different threads, which results in non-modular verification. However, a possibility might be to exploit this, by using constraints to specify the relies and guarantees that are necessary for a rely-guarantee style of verification. Another possibility is to redefine the notion of constraint, so that it only relates visible states within the same thread. However, in that case, one needs to ensure that other threads cannot change the state of the object, so that the constraint can be violated. Again, it is future work to investigate how the notion of history constraint can be exploited in a multithreaded setting.

Further, another way to solve the problem of interference for method contracts is to use finer-grained contracts for individual statements. This allows one to verify that the behaviour of an individual method is as expected, even if it cannot be expressed as a method contract. To accomplish this, `assert` annotations can be used, but they do not offer enough expressiveness. Instead, we have found that block specifications (currently called `following_behavior` in JML[1]) are appropriate for this. A `following_behavior` block is like a usual JML method contract, except that it applies to a code block inside a method (see Figure 4.1,

[1] A recent proposal proposes to tag them with a `refining` keyword.

described in Section 4.4 below for an example). Method specifications that suffer from interference, because the protecting locks are released by the method, can be specified as a `following_behaviour` – within the block that is protected by the lock. This allows one to verify the behaviour of the individual method, and it gives documentation to somebody who wishes to use the class. However, the method specification cannot be used to verify a call to the corresponding method.

To correctly and precisely specify the behaviour of multithreaded programs, we also often need a way to express the value "a variable had at some point in time". Consider for example the following method (from [28]), which models optimistic concurrency control. In this method, we wish to update the shared variable `z` according to `z = f(z)`. However, as `f` is a long-running operation, for performance issues, the lock protecting `z` is released during the call to `f`. The method keeps a local copy of `z`, computes the new value (`f(z)`) and assigns it to `z`, if `z` has not changed in the mean time. If `z` has changed, the method loops.

```
int z; //@ monitors_for z <- l;

Lock l;

int f(int i);

void apply_f(){
    int x, fx;

    l.lock();
    x = z;
    l.unlock();

    while(true){
        fx = f(x);
        l.lock();
        if(x == z){
            z = fx;
            l.unlock();
            return;
        }
        l :  x = z;
        l.lock();
    }
}
```

For this method, intuitively one would like to specify a post-condition `z == \old(z)`. However, this would be incorrect, since `\old(z)` might denote the value of `z` a long time ago (not necessarily the value of `z` before the last call to `f`). Instead, we have to express that at some point during the execution of this method, the variable `z` contained a value z such that after the method has finished `z == f(`z`)`.

JML allows one to write an expression `\old(`*lab*`,` E`)`, denoting the value that expression E had when control last reached the statement label *lab* [48, §11.4.2]. Thus, a correct and appropriate post-condition for this method is `z == f(\old(l, z))` (provided that `\old(`*lab*`,` E`)` denotes the value of expression E when control *in the current thread* last reached the statement label *lab*). However, one might argue that it actually contains too much information: all one wishes to specify is that `z` is equal to `f` applied to a value contained in `z` at some point in time. We will further study whether we can use (an abstraction of) the labelled `\old` expression to denote such expressions in method specifications. It is also unclear whether such method specifications only serve to verify the method implementation, and as documentation for the user, or whether they can also be used to verify method calls.

Finally, we would like to mention one construct that is provided by JML, explicitly to support multithreading: the `when` clause. We propose a slightly different semantics for this keyword, compared to the one given in the JML Reference Manual [48, §9.9.8] (described in more detail in [62]).

`when/commit` (from [48, §9.9.8])

- *Syntax* The `when` clause is a method specification clause, while the `commit` annotation appears in the method's body. The default `when` clause is `true` for a heavyweight specification, and `\not_specified` for a lightweight specification. The default position for the `commit` point is before each of the method's return points. If the `when` clause appears in different specification cases, it must always contain the same predicate, otherwise a specification error is reported.
- *Meaning* This pair of keywords (adapted from Lerner [49]) specifies that the method might block (at some point in its body) unless the predicate specified in the `when` clause is satisfied (typically, because of an action by another thread). The `commit` annotation indicates the point where the predicate specified in the `when` clause should hold.
- *Use* The predicate specified in the `when` clause can be assumed at the `commit` point to establish the post-condition of the method (provided the method can be verified sequentially after the `commit` point).
- *Verification* It has to be shown that the predicate specified in the `when` clause indeed holds, when the thread can resume execution. This will require a global analysis of the behaviour of the other threads, in order to know under which conditions they woke up the blocked thread.

So far, we have not studied further how to exploit the `when` clause.

4.3 Differences With Other Language Proposals

4.3.1 The Spex-JML Project

As part of the Spex-JML project, Rodríguez et al. [62] propose an extension of JML with concepts for multithreading. Our work has been highly inspired by their proposal, but there are some significant differences, that we will discuss here.

Rodríguez et al. focused on atomicity and independence, whereas we use contract-atomicity and contract-independence. First, contract-atomicity fits well in a framework based on JML specifications because it allows to reason in a modular and sound way. Second, it permits to detect erroneous specifications, such as contracts concerning objects that are not properly protected. Although it may seem naive, this is an important point because it is notoriously difficult to write well-defined specifications in a multithreaded environment.

Our definition of `when` and `commit` is slightly different from their proposal (and the description given in the JML Reference Manual [48]). First, we do not impose the method to be atomic between the commit point and the method's `return` statement. When using contract-atomicity this is no longer useful, because the definition of contract-atomicity is not based on reduction. Thus, we accept a larger number of programs without losing the usefulness of the annotations. Second, the point where the method may block is different. Indeed, Rodrìguez et al. state that the method blocks whenever it is a called in a state where the predicate in the `when` clause does not hold. However, the condition that makes the method block may be established *between* the point where execution of the method starts and the point where this condition is tested by the method. Therefore, we just specify that a method may block unless the predicate in the `when` clause is satisfied, but we do not require that this is decided immediately at the beginning of the method.

We do not support the `locks` keyword, because its semantics and its usefulness were unclear. According to Rodríguez et al., a method annotated with `locks l1, ..., ln` acquires and releases all the locks in this list when executing. However, it was not clear whether this means that all these locks will be held

simultaneously at some point, or throughout the whole method, or whether it would just mean that the method will acquire and release all locks in the given list without any further restriction. In this case, this method below would correctly satisfy its `locks` clause.

```
//@ locks l1, ..., ln;
void method(){
  synchronized(l1) { .. }
  ..
  synchronized(ln) { .. }
}
```

In any case, it was unclear to us how the `locks` specification could help verification of an application.

The keywords `lock_protected` and `thread_local`, proposed by Rodríguez et al., are subsumed by the keyword `shared`. The annotation `lock_protected(o)` (where `o` is a method parameter) is defined as: "`o` is access-protected by some nonempty set of locks and all of those locks are held by the current thread". However, the semantics of "access-protected" is unclear: it could mean that writes within `o` are forbidden or that all objects reachable through `o` cannot be written by other threads. The semantics of `shared` with a non-empty locking policy corresponds to the first possibility, while the second possibility can be obtained by adding `shared` annotations within all sub-objects of `o`.

Moreover, `thread_local` is used to state that an object is not shared between different threads. This can be captured by an annotation `shared(1,0)<1,0>`, meaning that only one thread at a time may execute within the object annotated. Contrary to `thread_local`, we provide a way to check the correctness of our annotation.

Finally, the keyword `thread_safe`, defined as `thread_local` $\vee$ `lock_protected` can be encoded by `shared(1,0)<1,0>` or an appropriate policy (i.e., an annotation of the form `shared(x,y)<w,z>{l1, ..., ln}`). However, it is not possible to express both behaviours in a single `shared` annotation.

Finally, as the rule forbidding synchronisation on objects inheriting from `Lock` makes the keyword `locked_if` useless, we do not consider it here.

4.3.2 The Spec# Project

The Spec# project [6] aims at the development of a specification language and verification approach for C#. Within this project, interesting work on an extension that allows static verification of concurrent object-oriented programming has been done [40, 42]. We will refer to this as the Boogie methodology here.

The focus of the Boogie methodology is proving the preservation of invariants, rather than detailed functional properties; yet, functional properties are of course needed to establish the preservation of invariants.

The Boogie approach uses a simple system for thread ownership, i.e., for controlling shared access to objects between threads. It distinguishes between thread-local and shared objects. A newly created object is thread-local, and it remains thread-local until a reference to it is passed to another thread, leading it to become shared; once shared, it remains shared forever, and there is no way to retrieve exclusive ownership in one thread. Note that our system for thread ownership is more expressive, as it allows more fine-grained access restriction to be given per thread.

To ensure thread-safety of method contracts, a method contract can only talk about thread-local objects, immutable objects, and shared objects (or, more generally, shared state) for which the corresponding locks are held (i.e., which are 'lock-protected').

A special `havoc` statement in the intermediate language can be used to express the possibility of unknown side-effects on the heap, for instance interference by other threads.

The Boogie approach does not rely on a separate checker to ensure the absence of data races. Absence of data races has to be proven, as part of any code verification process. To do this, an association between a state and the locks protecting the access to that state has to be specified, as in our approach, using the `monitors` clause. Also, the ownership type system tells which state belongs to an object. The approach uses an ownership type systems, with `rep` and `peer` annotations.

Given this, we are also interested in verifying properties of existing code. Using a separate checker for race conditions, as we propose, can have the effect of reducing the verification burden, especially if we are only interested in verifying relatively weak properties of existing code.

The Boogie methodology does require a particular coding style. One aspect of this is that it requires the use of explicit `pack` and `unpack` statements by the programmer to specify the code region during which an objects invariant may be broken. It is not clear if this programming style would be compatible with typical MIDlets.

The approach assumes that *all* the code has been verified, i.e., that all code meets the obligations that Spec# imposes. So it does not allow for any hostile code that might break Spec# rules. It is not immediately clear how easy it would be to relax this assumption. In our approach we do want to be sound in the presence of malicious code, possibly even malicious code that contains data races. (For instance, in defining a notion of immutable object, we found that the possibility of code which does not adhere to the rules of the ownership type system we use causes a considerable complication.)

One interesting aspect of the Boogie approach is the use of an intermediate language, BoogiePL, for programs with proof obligations. BoogiePL is a simple imperative language, extended with `assert` and `assume` statements to express proof obligations. It is not object-oriented, and uses an external axiomatisation of the heap. The semantics of object invariants, the ownership type system, the notion of thread-local object, and the notion of a lock-set – the set of locks owned by the current thread – are all expressed in BoogiePL, typically through the use of additional ghost fields. An advantage is that the issue of verifying BoogiePL programs is independent of the (possibly complicated) semantics of these features. This allows a modular set-up of the verification tool chain, with one component for the translation of Spec# to BoogiePL, defining the semantics of Spec#, and another component for the generation of verification conditions from BoogiePL. Such modularity is not only practically useful in tools, but also conceptually in coming to grips with the possibly complicated semantics of some notions in the higher-level specification language. Moreover, directly editing BoogiePL programs allows experimentation with (the semantics of) features in the higher level specification language.

It also means that BoogiePL can be used as intermediate language for other approaches, incl. ours. But note that for some notions, expressing their semantics in BoogiePL might be difficult or impossible; an example might be the notion of contract-atomicity.

4.4 Example

Finally, we finish this chapter with a typical example specification, illustrating the concepts and keywords introduced above. Figure 4.1 presents an example taken from Rodríguez et al. (see Figure 8 in their paper [62]) which highlights the difference between our approaches. This example shows the implementation of the `take` method which is used to retrieve an element from a concurrent container. The field `items` is used to store elements and `takeIndex` indicates the index where the next object is going to be retrieved.

First, the most notable difference with Rodríguez et al. is that `take` is specified as contract-atomic, which means that its contract can be used for modular verification. That was not possible with the approach of Rodríguez et al. where `take` was specified as atomic.

Second, the contract of `take` has a weaker post-condition in our framework. Indeed, in their paper, Rodríguez et al. put our `following_behavior` block in `take`'s method's specification. However, we believe that this is not sound, since, for example, the original post-condition `ensures \result == \old(items[takeIndex])` relies on `items` and `takeIndex` which can be concurrently modified before the method returns (because `lock` is released before the method returns). Indeed, if other threads retrieve elements from the container between `take` returning and the client resuming, `takeIndex` may not represent anymore the index where the object returned has been taken, thus violating the post-condition. This problem with the method specification is found immediately when verifying whether the method is contract-atomic.

Third, the various `shared` annotations make clear how objects are shared between threads. The annotation `shared(1,0)<1,0>{lock}` on the array `items` indicates that the array can be modified only through

```
class ArrayBlockingQueue {

    /*@ shared(1,0)<1,0>{lock} shared(0,0)<1,0>[*] @*/ private final E[] items;
    private transient int takeIndex;
    //@ monitors_for takeIndex <- lock;

    private final ReentrantLock lock;
    private final Condition notEmpty;
    private int count;
    //@ monitors_for count <- lock;

    /*@ when count != 0;
      @ assignable items[takeIndex], takeIndex, count;
      @ ensures \result != null;
      @*/
    public /*@ contract_atomic shared(1,0)<1,0> @*/ E take() throws InterruptedException {
        lock.lockInterruptibly();
        try {
            /*@ following_behavior
              @ ensures x == \old(items[takeIndex]) && count == \old(count)-1
              @         takeIndex == (\old(takeIndex)+1) % items.length;
              @*/
            {
                try {
                    while (count == 0)
                        notEmpty.await();
                }
                catch (InterruptedException ie) {
                    notEmpty.signal();
                    throw ie;
                }
                /*@ commit @*/ E x = extract();
            }
            return x;
        }
        finally { lock.unlock(); }
    }

    /*@ requires \lockset.has(lock);
      @ assignable items[takeIndex], takeIndex, count;
      @ ensures \result != null && \result == \old(items[takeIndex]) &&
      @         takeIndex == (\old(takeIndex)+1) % items.length &&
      @         count == \old(count)-1;
      @*/
    public /*@ independent shared(1,0)<1,0> @*/ E extract(){ .. }
}
```

Figure 4.1: Method `take`

this reference, i.e., no other thread can concurrently access it and furthermore, no assignment within `items` may occur without holding `lock`. In the original proposal, the array `items` was protected only with a `monitors_for` annotation. But this only protects the reference to the array, and not the elements in the array. If the elements in the array not protected by `lock`, the whole implementation is unsound.

In addition, the second annotation for the array specifies thread ownership properties for the elements in the array. The annotation `shared(0,0)<1,0>[*]` indicates that it is forbidden to change any of the objects

stored in `items`.

Finally, the annotation `shared(1,0)<1,0>` on the return type of method `take` indicates how an object returned by `take` is shared, i.e., the thread retrieving it has all rights on it. All these behaviours were left implicit or unspecified (and unchecked) in the original specification.

Chapter 5

Example Verifications

This chapter presents two examples to demonstrate the use of the notions developed in Chapter 3 and Chapter 4. The example in Section 5.1 shows how thread ownership and contract-atomicity can be used to verify a typical worker-thread pattern application. The example in Section 5.2 combines immutability and contract-atomicity to verify an instance of the copy-on-write pattern. It is future work to implement all the techniques used, here we informally describe how the notions defined before permit to verify mainstream multithreaded programs.

5.1 Contract-atomicity and Locality

The example in Figure 5.1 demonstrates how contract-atomicity and locality can be used to show that a method can be verified sequentially. Method `findInt` is designed to check whether an `Integer x` occurs in the blue list (`bl`) or in the red list (`rl`). To increase performance, this method creates a thread (whose code is visible in Figure 5.2) to inspect the blue list and another thread to inspect the red one, and it returns as soon as one thread has found the appropriate element. Method `findInt` returns `null` if the `Integer x` does not appear in `bl` or `rl`; in order to know if instances of `MyThread` have terminated, method `isOver` is used. The algorithm performing the search is located in the `run` method of `MyThread`. It is important to note that this class has been designed for local use, as it does not perform anything related with concurrency. In fact, it is an instantiation of the worker pattern from Doug Lea [47]. The worker class (here `MyThread`) does not know what the rest of the program is doing: just performs its task and stops. Indeed, as indicated by the various `unvarying` annotations in the class `MyThread`, it only assumes that the rest of the program does not concurrently access the used data. In order to show that `findInt` is contract-atomic, we have to show that the evaluation of its contract only relies on stable variables. Note that all contracts appearing in this example are well-formed, indicating that the programmer has written relevant specifications.

Second, we show that the annotations related to locality are correct. Indeed, in the body of `findInt`, several data exchanges occur, and we must check that they correctly respect both the `shared` annotations of `findInt` and the annotations in `MyThread`.

First, we see that two threads (because of `MyThread`'s class annotation `shared<1,1>`) may execute simultaneously within instances of `MyThread` class: one with the right to write (obviously, it is the object `MyThread` itself), and one with only the right to read (in our example, this will be the thread executing `findInt`, also called the main thread). As only one thread can execute in write mode in any instance of `MyThread`, we will be able to analyse this class sequentially. In order to do that, we must show that any execution of `findInt` respects `MyThread`'s class annotation `shared<1,1>`. Below, we informally explain how our draft type-system to check locality operates on `findInt`.

At point 1 (indicated in comments in the code of `findInt`), `t1` and `t2` have partial capacity `shared(1,1)` because the corresponding threads have not been started, and `bl` and `rl` have partial capacity `shared(0,1)` as indicated at the parameter declaration. After the two calls to `start`, partial capacities of `t1` and `t2` have changed to `shared(0,1)`, because the main thread has only the right to execute in read-only mode within

```
//@ assignable \nothing;
//@ requires bl != null && rl != null && x != null;
//@ ensures (bl.contains(x) || rl.contains(x)) ==> \result.equals(x);
Integer findInt(/*@ shared(0,1)<0,1> @*/ Vector<Integer> bl,
                /*@ shared(0,1)<0,1> @*/ Vector<Integer> rl,
                /*@ shared(0,1)<0,1> @*/ Integer x){

    Integer r = null;

    MyThread t1 = new MyThread(bl, x);
    MyThread t2 = new MyThread(rl, x);  // 1

    t1.start();
    t2.start();                         // 2

    while(true){
        if(t1.isOver()){                // 3
            r = t1.getResult();
            if(r != null)
                return r;
        }
        if(t2.isOver()){                // 4
            r = t2.getResult();
            if(r != null)
                return r;
        }

        if(t1.isOver() && t2.isOver()){ // 5
            r = t1.getResult();
            if(r != null)
                return r;
            r = t2.getResult();
            if(r != null)
                return r;

            return null;
        }
    }
}
```

Figure 5.1: `findInt` method

`t1` or `t2`. Partial capacities of `bl` and `rl` are updated to `shared(0,0)`, because, when they have been passed to `t1` and `t2`, their partial capacities have been "consumed", i.e., the main thread should not call methods or write fields of `bl` and `rl` anymore.

Until point 3, the main thread has only executed a pure method (`isOver`) within `t1`, thus the program is fine. If the main thread enters the `then` branch of the conditional, the partial capacity of `t1` is updated to `shared(1,1)` again, because we know that `t1` has stopped (this is due to how `MyThread` is implemented). That is why the call to `getResult`, even if it is a non-pure method, is fine: as `t1` is stopped, the execution of `getResult` does not violate `MyThread`'s class annotation `shared<1,1>`.

If the first conditional test is false, the same reasoning can be applied at point 4 and later at point 5 to show that, if the branch of a conditional is entered, the statements encountered can be type-checked. Note that, by point 2, partial capacities of `bl` and `rl` (`shared(0,0)`) have not been violated because the main thread has not used them. Finally, if no conditional is entered and the program loops, the same reasoning

```
/*@ shared<1,1> @*/ class MyThread extends Thread {

    /*@ unvarying<1> @*/ Vector<Integer> l;
    /*@ unvarying<1> @*/ Integer x;
    volatile private boolean isOver = false;
    volatile Integer result = null;

    //@ requires l != null && x != null;
    MyThread(/*@ unvarying<1> @*/Vector<Integer> l,/*@ unvarying<1> @*/ Integer x){
        this.l = l;
        this.x = x;
    }

    //@ also
    //@   assignable result, isOver;
    //@   ensures isOver() && (l.contains(x) ==> result.equals(x));
    public void run(){
        Iterator<Integer> i = l.iterator();

        while(i.hasNext()){
            Integer j = i.next();
            if(j.intValue() == x.intValue()){
                result = j;
                break;
            }
        }

        isOver = true;
    }

    //@ assignable result;
    //@ ensures isOver() ==> (l.contains(x) ==> \result.equals(x));
    public Integer getResult(){
        Integer r = result;
        result = null;
        return r;
    }

    //@ ensures \result == isOver;
    public /*@ pure @*/ boolean isOver(){
        return isOver;
    }

}
```

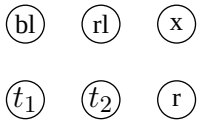
Figure 5.2: class `MyThread`

applies over and over again, so that the program can be type-checked *w.r.t.* localities. Note that this allows to verify the `MyThread` class sequentially because we have the property that no threads can concurrently write objects used in this class (thanks to the different `shared(0,1)<0,1>` annotations on fields `l` and `x` of class `MyThread`).

Now that we know localities are fine, we explain how the algorithm for contract-atomicity can be used to show that `findInt` can be analysed sequentially. As in Section 3.1, we denote stable variables in white circles (resp. unstable ones by grey circles)

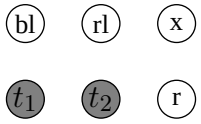
At point 1, `t1` and `t2` have been created but they have not been started, thus they are stable. Further,

`bl`, `tl` and `x` have the same partial capacity (`(1,1)`) as their complete capacity (`<1,1>`), so that the main thread is the only one having access to these three objects, which are consequently stable. Finally `r` is also stable, because it has just been created.



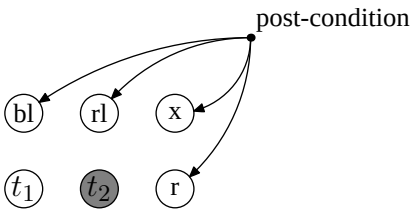
graph 1

However, after `t1` and `t2` have been started, they are no longer stable. By looking at the `MyThread` constructor, the algorithm deduces that the actual parameters (`bl`, `rl` and `x`) are still stable because no one can write to them.



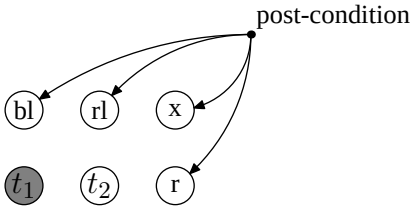
graph 2

If the `then` branch of the conditional at point 3 is entered, we know that `t1` is stopped, thus the call to `getResult` is secured and the assignment to `r` is secured. That is why, if the method returns by executing the conditional at point 3, the graph is as follows.

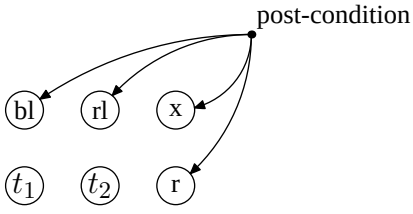


graph 3

We can see that the post-condition relies on `bl`, `rl`, `x` and `r`, which are all stable (this is represented by the different arcs). Similarly, if the `then` branch of the conditional at point 4 is entered, we are in almost the same case, except that now `t2` is stable (because it has stopped) while `t1` is not (as it keeps running). Finally, if the `then` branch of the conditional at point 5 is entered, both threads have stopped.



graph 4



graph 5

In all cases the evaluation of the post-condition solely depends on stable variables and, in case the method loops, the reasoning is the same. Thus, the method is contract-atomic and can be verified sequentially.

5.2 Lifting a Sequential Class to a Concurrent One

The example below illustrates how our techniques can be used to lift a class designed for sequential use to a concurrent environment. Essentially, we use contract-atomicity to show that reasoning with method contracts, even in the presence of interleavings, is sound.

Our example is based on an example from Doug Lea's book on concurrent programming in Java [47]. Lea uses this example to explain the copy-on-write technique. It is a simple implementation of a `Point` class that does not directly store the two coordinates in mutable fields, but instead stores the coordinates in an `ImmutablePoint` object. When a `Point` is moved, a new `ImmutablePoint` is created. We have added an invariant to Lea's example, which expresses the fact that a point must be in the lower right triangle of the plain, i.e., its `x`-coordinate must be greater than its `y`-coordinate. The following implementation of `Point` is meant to be used in sequential programs only. We have verified it with ESC/Java.

```
class Point {
  private /*@ non_null @*/ ImmutablePoint loc;

  //@ requires x >= y;
  public Point(int x, int y) {
    loc = new ImmutablePoint(x,y);
  }

  public ImmutablePoint location() { return loc; }

  protected void updateLoc (/*@ non_null @*/ImmutablePoint newLoc) {
    loc = newLoc;
  }

  //@ requires x >= y;
  public void moveTo(int x, int y) {
    updateLoc(new ImmutablePoint(x, y));
  }

  //@ requires 0 <= delta;
  public void shiftX(int delta) {
    updateLoc(new ImmutablePoint(loc.x + delta, loc.y));
  }
}

class ImmutablePoint {
  final int x;
  final int y;

  //@ invariant x >= y;

  //@ requires initX >= initY;
  public ImmutablePoint(int initX, int initY) {
    x = initX;
    y = initY;
  }
}
```

In order to make `Point` thread-safe, we have to synchronise accesses to the mutable `loc`-field. Note that without synchronisation the object invariant or `ImmutablePoint` could get violated when methods `shiftX()` and `updateloc()` execute concurrently.

```
class Point {
    private /*@ non_null @*/ ImmutablePoint loc;

    //@ monitors_for loc <- this;

    //@ requires x >= y;
    public /*@ contract_atomic @*/ Point(int x, int y) {
        loc = new ImmutablePoint(x,y);
    }

    public /*@ contract_atomic @*/ ImmutablePoint location() { return loc; }

    /*@ contract_atomic @*/
    protected synchronized void updateLoc(/*@ non_null @*/ImmutablePoint newLoc) {
        loc = newLoc;
    }

    //@ requires x >= y;
    public /*@ contract_atomic @*/ void moveTo(int x, int y) {
        updateLoc(new ImmutablePoint(x, y));
    }

    //@ requires 0 <= delta;
    public /*@ contract_atomic @*/ synchronized void shiftX(int delta) {
        updateLoc(new ImmutablePoint(loc.x + delta, loc.y));
    }
}

/*@ immutable @*/ class ImmutablePoint {
    final int x;
    final int y;

    //@ invariant x >= y;

    //@ requires initX >= initY;
    public /*@ contract_atomic @*/ ImmutablePoint(int initX, int initY) {
        x = initX;
        y = initY;
    }
}
```

There are at least two ways of statically checking that all methods in this example are contract-atomic. The first one is based on the *dependency analysis* outlined in Section 3.1.2. The second one combines *atomicity type-checking* with verifying that other threads cannot interfere with method contracts and object invariants. We will now outline both techniques.

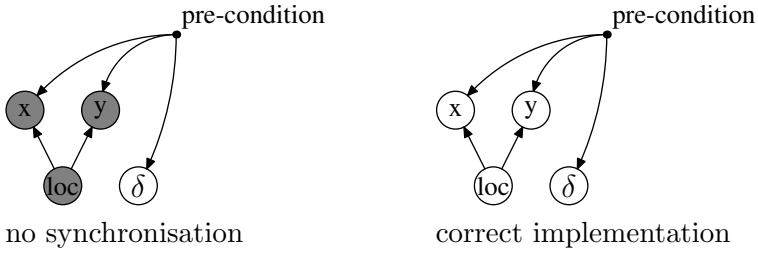


Figure 5.3: Contract-atomicity checking of `shiftX`

Showing contract-atomicity by dependency analysis. To show that all methods are contract-atomic, we first expand the method contracts by the object invariants, as it is done for sequential programs. For instance, the expanded contract for `updateLoc()` would be this:

```
//@ requires loc.x >= loc.y;
//@ requires x >= y;
//@ ensures loc.x >= loc.y;
void moveTo (int x, int y)
```

Now we use the dependency analysis from Section 3.1.2 to show that each method is indeed contract-atomic with respect to its contract. It is easy to show that `Points`'s constructor and methods `location()`, `updateLoc()` and `moveTo()` are contract-atomic, either because there is no property to verify or because they do not depend on the heap (they only use integers).

Checking that `shiftX()` is contract-atomic is less simple because we have to make sure that the call to `ImmutablePoint`'s constructor respects its pre-condition (even, this is the only proof obligation). However, without synchronisation, this pre-condition could get violated when methods `shiftX()` and `updateLoc()` execute concurrently. Indeed, if the `loc`-field is not protected, `shiftX` cannot be shown contract-atomic, because, when type-checking the statement `new ImmutablePoint(loc.x + delta, loc.y)`, the pre-condition of `ImmutablePoint`'s constructor depends on unstable values (as the graph on the left shows).

If synchronisation is used, `loc` is stable and the fact that class `ImmutablePoint` is immutable guarantees that `loc.x` and `loc.y` are stable as well. Then, our algorithm verifies that `shiftX()` is contract-atomic (as the graph on the right shows); consequently, the class `Point` can be safely used in a multithreaded environment because all its methods are contract-atomic.

Showing contract-atomicity by atomicity type-checking. Alternatively, we could prove contract-atomicity by combining a type-based atomicity analysis with interference freedom tests for method contracts and object invariants. The interference freedom tests have the same function as in Owicki-Gries- or Rely/Guarantee-style program logics, but a separate static atomicity analysis reduces the number of such tests. The idea to combine atomicity analysis and program verification is not new and has been proposed, for instance, in Rodríguez et al. [62].

Concretely, the following three conditions are sufficient for a method to be contract-atomic:

1. The methods is atomic. (Atomic Methods)
2. Its pre- and post-condition are stable against thread interference. (Stable Contracts)
3. Object invariants are stable against thread interference. (Stable Invariants)

We anticipate that sometimes we will have to deal with methods that are not contract-atomic. In such cases, we will have to do interference freedom test inside non-atomic methods in addition to (Stable Contracts) and (Stable Invariants).

In the example, showing (Stable Contracts) is simple because the method contracts only depend on integer values. In more complicated examples, method contracts depend on the heap and a dependency analysis as described in Section 3.1.2 is needed. We point out that sometimes *history constraints* will be helpful for showing (Stable Contracts).

Showing (Stable Invariants) is trivial for so-called *strong invariants*. Strong invariants are invariants that never get broken, not even temporarily. Typical examples of strong invariants are object invariants of immutable objects. We anticipate that in concurrent programs strong invariants are considerably more important than in sequential programs.

Here is how we show the three conditions for contract-atomicity in this example:

1. (Atomic Methods) can be verified using Flanagan et al.'s atomicity type system [30].
2. (Stable Contracts) is obvious, because none of the method contracts depends on the heap.
3. (Stable Invariants) is also obvious, because the only invariant in this example is an invariant on an immutable object and, therefore, it is a *strong invariant*. Immutability of `ImmutablePoint` could be shown by our immutability type system [35], although for this simple example this may be an overkill; in fact, there is a simple reason for the immutability of an `ImmutablePoint`: it has a shallow state and all its fields are `final`.

Chapter 6

Conclusions and Future Work

6.1 Summary of Current Results

This document describes the intermediate results of Task 3.3 of the MOBIUS project on verification of multithreaded applications.

We first discuss the role of the Java Memory Model. The Java Memory Model guarantees that any execution of a program that is correctly synchronised is sequentially consistent (i.e., is equivalent to an execution described by an interleaving semantics). We are formalising the Java Memory Model in Coq, and proving the guarantee. We exploit this by defining further verification techniques only for correctly synchronised programs. A program is correctly synchronised if it does not contain data races. To be able to check this efficiently, we have revived RCC, a static race condition checker.

Further, we have identified several conditions that can help to achieve thread modular verification, i.e., sequential verification of the behaviour of a single thread. First, we define the notion of contract atomicity, meaning that other threads cannot influence whether a method respects its contract. Second, we exploit the notion of immutability: if an object is immutable, objects to it do not have to be synchronised. And finally, we propose an annotation system for thread ownership. The information that can be derived from these annotations can be used to infer other properties.

We also propose an extension of the JML specification language with concurrency-specific keywords. Part of these keywords are based on the conditions that we have identified for thread-modular verification; the other keywords allow one to express e.g., locking policies and guarantees.

Finally, we discuss several example verifications, that illustrate how our different techniques allow to verify non-trivial multithreaded applications. The verifications are now done manually, but it is foreseen to implement the developed techniques as part of the MOBIUS tool set.

6.2 Plans

For the remainder of this task, we will continue the work described in this deliverable. Concretely, this means that we will further develop the following topics.

- Establishing a formal connection between the Java Memory Model description, where program behaviour is defined in terms of actions and orders on these actions, and BicolanoMT, where program behaviour is defined in terms of state traces.
- Developing a verification technique to check for contract atomicity, together with a formal correctness proof.
- Developing a verification technique technique to verify the thread ownership annotation system, together with a formal correctness proof.

- Describing a precise semantics for the new concurrency-specific keywords that extend JML (if not already done), and ensuring that we have appropriate verification techniques for all proposed keywords.
- Studying the impact of multithreading on visible state invariants, by distinguishing between invariants that are *never* broken (so-called *strong invariants*) and invariants that may temporarily be broken.
- Studying the impact of multithreading on the notion of history constraints. We also would like to explore whether constraints could be used to support reasoning in rely-guarantee style, by using them to express the rely and guarantee for a class.
- Studying and solving the possible (unwanted) interactions between the different extensions of JML that we propose.
- Implementing and integrating the developed verification techniques as part of the MOBIUS tool set.

We have also defined several topics that might be interesting to address further, but where we are not sure that we will have the time and possibility to do this within the context of the MOBIUS project.

- Studying the impact of benign race conditions. Benign race conditions are said to happen when two threads might try to write concurrently the *same* value to a location. This could for example happen if two threads are simultaneously computing and storing the hash key for an object. The outcome of the computation should always be the same, and thus the race condition should not be observable to the user. We would be interested in studying whether this is actually the case for the Java Memory Model.
- Formalising the procedure implemented by RCC to check for data race freeness and to prove soundness of the result of the checker *w.r.t.* the BicolanoMT semantics.
- We are also interested in studying the modularity of the Java Memory Model. Can we separate a program execution in two parts: one with and one without race conditions, and in this case, can we assume that the executions of the data race free part of the program are sequentially consistent? This property would be crucial to ensure that an application can work correctly in an untrusted context, where there might be data races.
- We are also thinking about a formal definition of the out-of-thin-air guarantee, and whether this is a property that can actually be formally proven for the Java Memory Model.
- There exists some work about slicing methods based on their specification [16]. We are interested in exploring the relation with contract atomicity: if a method is atomic *w.r.t.* its contract, would this then imply that the sliced method *w.r.t.* the contract is atomic in the classical sense?
- We would like to reviewing the various definitions of purity with the intent of refining the existing definitions to account for concurrency.
- In the literature, we have found several examples where we cannot specify its intended behaviour with (our extension of) JML. The method `alloc` in Figure 3.1 (on page 37) is one example of such a method.

  We have also encountered several examples where the "intuitive" post-condition might be broken immediately after the method has finished, because the appropriate locks are released. In such a case, one can give a block specification that specifies the intended behaviour of the method within the synchronised block. However, we can imagine that there are cases where the fact that the post-condition of the block specification actually was true at some point in time, might be useful information also for the caller of the method. For example, if the block specification stated that some element was added to some collection, it might be useful to know that the element once belonged to the collection. To be able to express that, we would have to extend JML with temporal operators.

Bibliography

[1] M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

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

[3] 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.

[4] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[5] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *International Symposium on Computer Architecture*, pages 2–14, 1990.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In Barthe et al. [7], pages 151–171.

[7] G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors. *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[8] J. Bloch. *Effective Java.* Addison-Wesley, 2001.

[9] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–230. ACM Press, November 2002.

[10] 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.

[11] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer-Verlag, 2001.

[12] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In *Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003.

[13] N. Cataño and M. Huisman. Chase: A static checker for JML's assignable clause. In *Verification, Model Checking and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40. Springer-Verlag, 2003.

[14] B.-C. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Programming Languages Design and Implementation*. ACM Press, 2000.

[15] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, 2003.

[16] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *Symposium on Applied Computing*, pages 605–609. ACM Press, 2001.

[17] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–310. ACM Press, 2002.

[18] D. G. Clarke, J. Noble, and J. Potter. Simple Ownership Types for Object Containment. In J. Lindskov Knudsen, editor, *European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 53–76. Springer-Verlag, 2001.

[19] 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 Barthe et al. [7], pages 108–128.

[20] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, March 2004. `http://coq.inria.fr/doc/main.html`.

[21] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical report, System Research Center, 1998.

[22] W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In E. Ernst, editor, *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 28 – 53. Springer-Verlag, 2007.

[23] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[24] P. C. Diniz and M. C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, 1998.

[25] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *International Symposium on Computer Architecture*, pages 434–442, 1986.

[26] C. Flanagan. Verifying commit-atomicity using model-checking. In *SPIN Workshop on Model Checking of Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, April 2004.

[27] 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.

[28] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(4):275–291, 2005.

[29] C. Flanagan and S.N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Principles of Programming Languages*, pages 256–267, New York, NY, USA, 2004. ACM Press.

[30] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Programming Languages Design and Implementation*, volume 38 of *ACM SIGPLAN Notices*, pages 338–349. ACM Press, May 2003.

[31] C. Flanagan and S. Qadeer. Types for atomicity. In *Types in Language Design and Implementation*. ACM Press, 2003.

[32] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26, 1990.

[33] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, 1989.

[34] A. Greenhouse and J. Boyland. An object-oriented effects system. In R. Guerraoui, editor, *European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229. Springer-Verlag, 1999.

[35] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In R. De Nicola, editor, *ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 2007.

[36] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI*, pages 175–190, 2004.

[37] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Static Analysis Symposium*. Springer-Verlag, 1998.

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

[39] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 34(10), pages 132–146, 1999.

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

[41] 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.

[42] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *International Conference on Formal Engineering Methods*, pages 420–439, 2006.

[43] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[44] JSR 133: Java memory model and thread specification, 2004.

[45] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[46] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Computer*, 28(9):690–691, 1979.

[47] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Boston, MA, USA, 1996.

[48] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[49] R. A. Lerner. *Specifying objects of concurrent systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[50] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification. Second Edition*. Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.

[51] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[52] J. Manson. *The Java Memory Model*. PhD thesis, Faculty of the Graduate School of the University of Maryland, 2004.

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

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

[55] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[56] R. H. B. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[57] P. Nienaltowski and B. Meyer. Contracts for concurrency. In *Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages*, July 2006.

[58] L. P. Nieto. The Rely-Guarantee method in Isabelle/HOL. In P. Degano, editor, *European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, 2003.

[59] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975.

[60] D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inria.fr/bicolano`. Summary appears in [54], 2006.

[61] W. Pugh. Fixing the Java memory model. In *Java Grande*, pages 89–98, 1999.

[62] E. Rodríguez, M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In A. P. Black, editor, *European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer-Verlag, July 2005.

[63] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Verification, Model Checking and Abstract Interpretation*, pages 199–215, 2005.

[64] F. Spoto and E. Poll. Static analysis for JML's assignable clauses. In G. Ghelli, editor, *ACM Workshop on Foundations of Object-Oriented Languages*. ACM Press, January 2003. Available at `www.sci.univr.it/~spoto/papers.html`.

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

[66] J. Thornley. *A Parallel Programming Model with Sequential Semantics*. PhD thesis, California Institute of Technology, 1996. Available as Caltech technical report CS-TR-96-12.

[67] J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.

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

[69] T. Zhao, J. Palsberg, and J. Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, January 2006.