An Assertional Proof System for Multithreaded Java
– Theory and Tool Support –

Erika Ábrahám

An Assertional Proof System for Multithreaded Java – Theory and Tool Support –

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D. D. Breimer,
hoogleraar in de Faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op donderdag 20 januari 2005
te klokke 14.15 uur

door

Erika Ábrahám
geboren te Szeged (Hongarije)
in 1970

Promotiecommissie

Promotores:	Prof. Dr. J. N. Kok
	Prof. Dr. W.-P. de Roever *Christian-Albrechts-University, Kiel*
Copromotores:	Dr. F. S. de Boer
	Dr. M. Steffen *Christian-Albrechts-University, Kiel*
Referent:	Prof. Dr. M. Wirsing *Ludwig-Maximilians-University, Munich*
Overige leden:	Prof. Dr. S. M. Verduyn Lunel
	Prof. Dr. B. Jacobs *Katholieke Universiteit Nijmegen*
	Prof. Dr. E.-R. Olderog *University of Oldenburg*

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2). Work carried out at the Christian-Albrechts-University, Kiel.

An Assertional Proof System for Multithreaded Java
– Theory and Tool Support –
Erika Ábrahám.
Thesis Universiteit Leiden. - With ref.

ISBN 90-9018908-4
IPA Dissertation Series 2005-01
Cover: "Java Island", designed by Judith Mumm

"When I am working on a problem, I never think about beauty. I think only of how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong."

– Buckminster Fuller

Preface

Now the work is done, and I would like to thank all people who helped me during my Ph.D. research.

I thank my professor Willem-Paul de Roever. He has provided and maintained a stimulating and challenging scientific environment in which my research could be successfully carried out. He did not only take care of the organization and research coordination between the various working groups of our projects, but contributed also to the research, and adapted the working conditions to my personal needs - arranging for me the possibility to move, as a member of his group, while supported by the Deutsche Forschungsgemeinschaft (DFG), to David Basin's group at Freiburg. I would also like to express my gratitude to Gerit Sonntag for making my move possible.

I am grateful to Frank de Boer. He acted as research leader of our bilateral Mobi-J project, which enabled me to undertake this piece of research. The Centre for Mathematics and Computer Science (CWI) was host to many working meetings, providing a platform for fruitful discussions and improvements. Frank was always full of new ideas and always willing to discuss them with me, during the day, or even in the evening in the pub.

I thank Martin Steffen, my daily research leader at the University of Kiel, who was closely involved in everything, and who was always there when I needed his help. He guided me unselfishly without forcing me into any particular direction, letting me find my own path and follow my own interests, while always accompanying me in scientific matters. I've always enjoyed our discussions, the ups and downs of problem solving, and his way of teaching me to keep the overall view of what I am doing in my mind, while working out the details.

The original theme of my thesis was born during a lecture by Martin Wirsing, during which Willem-Paul realized that a Hoare-style proof theory for concurrent Java was now within reach of his team and could be made into the centre piece of what was to become the Mobi-J project.

The work presented in this thesis has been carried out in the context of the Dutch-German bilateral research project Mobi-J ("Assertional methods for mobile asynchronous channels in Java") having the partners CAU (Christian-Albrechts-University, Kiel), LIACS (Leiden Institute of Advanced Computer Science, Leiden), and CWI (Centrum voor Wiskunde en Informatica, Amsterdam). This project aims at the development of a programming environment which supports component-based design and assertional verification of concur-

rent Java programs. Frequent meetings between the working groups gave room for intensive discussions and new ideas. The FMCO'02 (First International Symposium on Formal Methods for Components and Objects), organized as part of the Mobi-J project, was a fruitful platform for exchanging ideas with other research groups working on related topics. I would like to thank the DFG and the Netherlands Organization for Scientific Research (NWO) for their financial support.

I thank David Basin for enabling me to be a guest at his chair. His support was not restricted to an official working place, only, but also offered me new possibilities to exchange ideas and to learn more about research in other areas. I am especially grateful for his remarks on my thesis and for teaching me the importance of not getting lost in theoretical results, by putting them into a scientific context, and pointing out their use and advantages.

I am grateful to Stefan Friedrich and all other people who read (parts of) this thesis and gave useful remarks.

I thank Ulrich Hannemann for all his support during the first two years of my research activities when I was analyzing hybrid systems. Work at Kiel would have been almost impossible without the support of Änne Straßner, who reliably solved all bureaucratic issues for the whole chair, and who always had time for smoking a cigarette with me. Dear Softtech groups in Freiburg and in Kiel, I am sorry for the occasional noise made by my children when they were playing on the floor...

I would like to thank all friends who helped me in everyday life, taking care of my children and giving me the power to persevere when I thought that I wouldn't make it... My special thanks go to our child minder Tante Waltraut. It would have been very hard to arrange life without her help and love. I thank my parents and family, who showed great understanding for my work. Finally, I thank all other people who helped me in some way or another during those last four years to keep my family together while I worked on the research described in this thesis.

Contents

Chapter 1

Introduction

Java [GJS96] is a widely used programming language. Its growing popularity, since its first release in 1995, parallels the growth of the Internet for the programming of which it was designed.

Java's history begins in late 1990, when Sun Microsystems initiated the Oak project. The goal was to design a technology that could integrate electronic consumer devices via the Internet with other computing devices using a standard programming language.

The *Java* platform is based on the power of networks and the idea that the same software should run on many different kinds of computers. During the last years, *Java* technology has rapidly grown in popularity because of its portability.

The syntax of the *Java* language is similar to the syntax of C and C^{++}. Therefore, it is both familiar to and easily learned by C and C^{++} programmers. However, some features of those languages —like pointers, the lack of automatic memory management, and multiple inheritance— were changed.

Java programs are compiled into bytecode, which can be interpreted by the Java Virtual Machine (*JVM*). Bytecode programs are platform-independent, i.e., they can be run on any platform to which a *JVM* has been ported. The *JVM* bytecode verifier checks *JVM* code for type consistency and other static properties.

Since the language is increasingly used in safety-critical applications, verification techniques for *Java* programs become increasingly important. *Java* has several interesting and challenging features like object-orientation, inheritance, and exception handling. Furthermore, *Java* integrates concurrency via its `Thread`-class, allowing for a multithreaded flow of control.

To reason about *safety* properties of multithreaded *Java* programs, this thesis introduces a tool-supported *assertional proof method* for a concurrent sublanguage of *Java*. The language includes dynamic object creation, object references with aliasing, method invocation, reentrant code, and, specifically, *concurrency* together with *Java*'s *monitor discipline*. The concurrency model includes shared-variable concurrency via instance variables, coordination via reentrant synchronization monitors, synchronous message passing, and dynamic thread creation. The results of this thesis are formulated for a *Java* sublanguage, but they can be adapted to other concurrent class-based object-oriented programming languages having similar features.

We illustrate our assertional proof system on a number of examples, which have been verified using the tool *Verger* (*VERification condition GEneratoR*). This tool takes a *Java* program together with its specification, a so-called proof outline, as input and generates the verification conditions which assure invariance of the specification. We use the theorem prover *PVS* [ORS92] to verify those conditions.

In [Lam94] Lamport asserts that "although these methods [basing on the Owicki-Gries approach] have been reasonably successful at verifying simple algorithms, they have been unsuccessful at verifying real programs. I do not know of a single case in which the Owicki-Gries approach has been used for the formal verification of code that was actually compiled, executed, and used. I do not expect the situation to improve any time soon. Real programming languages are too complicated for this type of language-based reasoning to work."

This thesis will not brake the wall, but it is a step in that direction. We formalize an assertional proof system for a real concurrent object-oriented language, develop tool support for the correct and complete generation of the verification conditions, and use the theorem prover *PVS* to prove the conditions interactively.

Though we apply the tool to several examples (see Section 9), we did not carry out any large case studies yet. Besides the extension of the proof system to further language features and the optimization of the tool support and the *PVS* implementation, such a case study belongs to the topics of interest in the Mobi-J project.

The verification process consists of three phases (see Figure 1.2): First the user has to *annotate* the given program with predicates which should hold during program execution when the flow of control reaches the annotated point. Afterwards, the proof system has to be applied to the annotated program, resulting in so-called *verification conditions*. These conditions assure that the annotation describes program execution correctly. Finally, the verification conditions must be *proven* using the theorem prover.

Our experience has shown that most of the user effort must be put into the specification of the annotation. The *Verger* tool takes care of the second phase, i.e., it automatically generates the verification conditions for an annotated program in the syntax of *PVS*. The third phase, the actual verification process within the theorem prover, is interactive. However, for our examples most of the conditions could be proven automatically, without user interaction, using the built-in proof strategies of *PVS*. Human interaction was needed mostly for the proof of properties whose formulation required quantifiers.

As a consequence, in the future we will concentrate to the development of further computer support for the first phase, specifying the invariant program properties. The *Verger* tool is already able to automatically generate the weakest preconditions of assignments, if required. That means, the user only needs to define the postcondition of an assignment, and let the tool generate the precondition automatically. However, due to shared-variable concurrency, these predicates are not always invariant.

It would be also interesting to restrict the logic to a decidable subset for which fully automatic verification is possible within the theorem prover.

We expect that the above observations would hold also for larger case studies. Though for larger programs more verification conditions are generated, their proofs are independent of each other. Thus the program size influences the number but not the complexity of single conditions.

In the following we informally describe the contents of this thesis in Section 1.1. We discuss *Java*'s monitor concept in Section 1.2, and give a short introduction to Hoare logic in Section 1.3. Finally, Section 1.4 gives an overview of the remainder of the work.

Related work is discussed at the end of each chapter. A wider field of research topics on *Java* like, for example, the semantics of bytecode, type checking, model checking, etc., is discussed in e.g. [HM01].

1.1 The proof system

As mentioned above, in this thesis we formulate an assertional proof system for a multithreaded sublanguage of *Java*, excluding inheritance, subtyping, and exception handling.

To transparently describe the proof system, we present it incrementally in three stages: We start with a proof method for a *sequential* sublanguage of *Java*, allowing for dynamic object creation and method invocation. This first stage shows how to handle activities of a single process, i.e., a single *thread* of execution. In the second step we additionally allow dynamic thread creation, leading to *multithreaded* execution. The corresponding proof system extends the one for the sequential case with conditions handling dynamic thread creation and the new interleaving aspects. Finally, we integrate *Java*'s *monitor synchronization* mechanism. Monitor synchronization allows the implementation of mutual exclusion within objects.

This incremental development shows how the proof system can be extended stepwise to deal with additional features of the programming language. We are currently working on the integration of exception handling [ÁdBdRS04b]. Further extensions by, for example, inheritance and subtyping are topics for future work (see Section 8) [PdB03].

A program is given by a set of classes, where each class defines its own methods and instance variables. Concurrently executing threads can communicate using the shared instance variables of class instances, i.e., objects.

To support a clean interface between internal and external object behavior, we exclude qualified references $o.x$ referring to instance variables x of objects o. I.e., the values of instance variables of an object can be accessed and modified only within the object. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only, but not across object boundaries. Of course, each program containing qualified references can be transformed into another one without qualified references, by defining for each class special methods for reading and writing the values of instance variables, and replacing each qualified reference by a method call invoking those special methods. For example, the classes

```
public class C1{
    public void m(C2 o, int n){
        int u;

        u = n;
        o.x = u;
    }
}

public class C2{
    public int x;
}
```

can be transformed into the following ones without qualified references:

```
public class C1{
    public void m(C2 o, int n){
        int u;

        u = n;
        o.set_x(u);
    }
}

public class C2{
    private int x;

    public void set_x(int v){
        x = v;
    }
}
```

Why is it advantageous from a proof-theoretical point of view to exclude qualified references to instance variables? Without qualified references, properties of an object's state are automatically invariant under execution in other objects. For the above examples, assume that instances of the class `C2` have the invariant property that the value of x is non-negative. To prove this property, for the first example we have to show its invariance under execution in instances of both classes, since the assignment `o.x=u` executed in a `C1`-instance changes the state of the instance `o` of `C2`. In the second example, the given property of a `C2`-instance is independent of execution in other objects. Thus to show invariance we only have to take execution within the `C2`-instance into account.

In order to capture this modular program behavior, the assertional logic and the proof system are formulated at two levels, a local and a global one. The local assertion language describes the internal object behavior. A local assertion in the specification of a method m of an object o refers to the instance variables of o and to the local variables of m of o. Such a local assertion can state, for example, that the value of the integer instance variable x of o is positive, or that the local variables u and v of the same type have the same value.

The global behavior, including the communication topology of objects, is expressed in the global language. As in the Object Constraint Language (OCL) [WK99], properties of object-structures are described in terms of a navigation or dereferencing operator. A global assertion may state for example that the value of the instance variable x of an object o_1 equals the value of the instance variable x of another object o_2, or that the number of all existing, i.e., already created, objects is stored in the integer instance variable n of an object o_3.

As explained in the following sections, most of the verification conditions are formulated in the local assertion language, which is free from qualified references and from quantification over objects. Our experience has shown that proving local assertions using a theorem prover can be done with a large degree of automation. In most cases, user interaction is needed only for proving global conditions which contain quantification.

The assertional proof system is formulated in terms of *proof outlines* [OG76], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [Flo67, Hoa69]. To give a feeling of how an annotation looks like, assume the following partial[1] annotation for the previous example without qualified references, which defines local assertions $\{p\}$ attached to control points in the methods of the classes `C1` and `C2`. Additionally, the class `C2` has a so-called *class invariant* $x \geq 0$.

```
public class C1{
    public void m(C2 o, int n){
        int u;

        {n ≥ 0}
        u = n;
        {u ≥ 0}
        o.set_x(u);
    }
}

public class C2{
    private int x;
    {x ≥ 0}

    public void set_x(int v){
        {v ≥ 0}
        x = v;
    }
}
```

This annotation states that if during program execution control stays prior to the assignment `u = n` in method `m` of an instance of `C1` then $n \geq 0$ holds, and similarly for the method call `o.set_x(u)`, $u \geq 0$ is assumed to hold prior to its execution. The annotation in `C2` is similar, where the class invariant $x \geq 0$ is required to hold during the whole life cycle of `C2`-instances.

The satisfaction of the program properties specified by the assertions is guaranteed by the verification conditions of the proof system. The *initial correctness* conditions cover satisfaction of the program properties in the initial program configuration. The execution of a single method body in isolation is captured

[1] Control points which are not explicitly annotated get assigned the assertion `true`.

by standard *local correctness* conditions, using the local assertion language. Interference between concurrent method executions is covered by the *interference freedom test* [OG76, LG81], formulated also in the local language. It has especially to accommodate reentrant code and the specific synchronization mechanism. Possibly affecting more than one instance, method call and object creation is treated in the *cooperation test*, using the global language. Method calls can take place within a single object or between different objects. As these cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules in [AFdR80] and in [LG81] for CSP.

For the above example, assume that we would like to prove invariance of the annotation. Local correctness assures that if $n \geq 0$ holds prior to the execution of `u = n`, then $u \geq 0$ holds afterwards. Interference freedom takes care, for example, of the invariance of the class invariant: If the precondition $v \geq 0$ of the assignment `x = v` and the class invariant $x \geq 0$ hold prior to the execution of the assignment, then the class invariant is required to hold afterwards. Finally, the cooperation test requires that if the precondition $u \geq 0$ of the method call `o.set_x(u)` holds prior to the call, then the precondition $v \geq 0$ of the method body holds after invocation.

Our proof method is *modular* in the sense that it allows for separate interference freedom and cooperation tests (Figure 1.1). This modularity, which in practice simplifies correctness proofs considerably, is obtained by disallowing the assignment of the result of communication and object creation to instance variables. Clearly, such assignments can be avoided by additional assignments to fresh local variables and thus at the expense of new interleaving points. This restriction could be omitted, without loosing the mentioned modularity, but it would increase the complexity of the proof system (see Section 8.2).

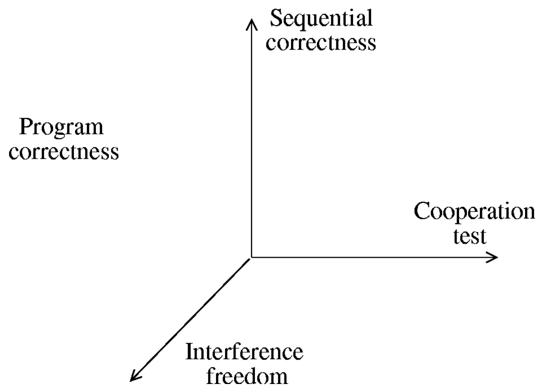


Figure 1.1: Modularity of the proof system

Thus we have three kinds of modularity:

- modularity of the programming language: a clean interface between internal and external object behavior,

- modularity of the logic: local and global assertions describe object-internal and object-external behavior, respectively, and
- modularity of the proof system: separate verification conditions for intra-object and inter-object computation.

Our modular proof system allows one to verify object-internal properties on the local level, independently of the context, using assumptions about the environment's communication properties. These assumptions are validated on the global level by the cooperation test. Consequently, if instances of a class are proven to satisfy some specification in a given context, then they will satisfy the specification also in all other contexts as far as the context satisfies the assumptions about the communication structure. That means, if a class of a program gets replaced by another one, we do not have to prove again the whole new program to be correct with respect to its specification: local proofs are reusable, only the global proofs must be redone.

Computer-support is given by the tool *Verger* (*VERification condition GEneratoR*), taking a proof outline as input and generating the verification conditions as output. We use the interactive theorem prover *PVS* to verify the conditions (cf. Figure 1.2). The verification conditions are generated by a Hoare

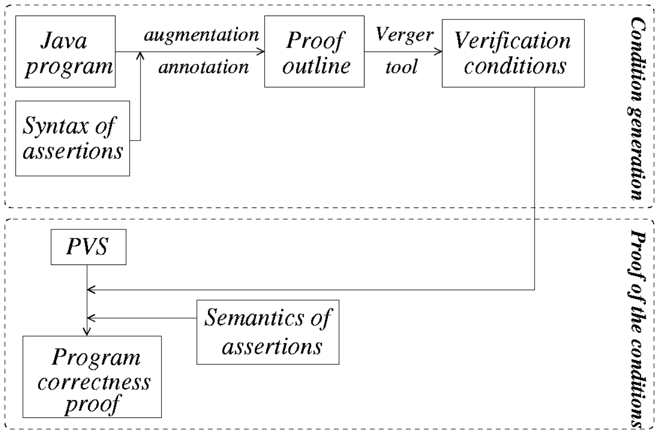


Figure 1.2: The verification process

logic which is based on a syntactic modeling of assignments by means of substitutions; the verification conditions are standard logical implications. Therefore, we only need to encode the semantics of the assertion language in *PVS*, instead of the semantics of assignments, whose encoding is needed for more semantically-oriented approaches based on the global store model [AL97, JKW03, vON02].

This thesis puts together uniformly and extends earlier results. America and de Boer [AdB90b] formulate the first time a cooperation test for an object-oriented language called SPOOL with synchronous message passing. In [ÁMdB00] we generalize this work to *Java* and extend it to concurrency, but without reentrant monitors. This generalization consists of an extension of the cooperation test to method calls and a definition of an interference freedom test. Reentrant monitor synchronization was incorporated in [ÁMdBdRS02c, ÁdBdRS03b]. An incremental description of the proof system, starting with a sequential language and stepwise adding additional language features, is given in [ÁMdBdRS02b]. In [ÁMdBdRS02b] we also introduce proof conditions for deadlock freedom. A more informal and intuitive discussion of the proof system with and without monitor synchronization can be found in the extended abstracts [ÁMdBdRS01] and [ÁdBdRS03d], respectively. Currently we are working on the incorporation of *Java*'s exception handling mechanism [ÁdBdRS04b]. We formalize the semantics of our programming language in a compositional manner in [ÁdBdRS04a].

The proof system and its application is explained in detail in technical reports [ÁMdBdRS02a, ÁMdBdRS02d, ÁdBdRS03a, ÁdBdRS03c], including also the corresponding soundness and relative completeness proofs.

This thesis integrates and extends the above results with additional examples illustrating annotation, augmentation, the application of the verification conditions, and how to prove deadlock freedom. The above papers formalize the verification conditions as standard Hoare triples. We define their formal semantics by means of substitutions. Finally, we describe the tool support, which is not published yet, and give some examples illustrating its use.

Our work defines the first sound and relatively complete tool-supported assertional proof method for a multithreaded sublanguage of *Java* including its monitor discipline. The main contribution of this thesis lies on *concurrency* and *monitors*. Related work, which is discussed at the end of each chapter, deals mostly with sequential languages.

1.2 Monitors

Monitors, first outlined in Hoare's article [Hoa74], offer a special mechanism of concurrency control used to simplify the implementation of mutual exclusion. A monitor consists of some local data together with some procedures and functions to acquire and release resources. From [Hoa74] we quote: "The procedures of a monitor are common to all running programs, in the sense that any program may at any time attempt to call such a procedure. However, it is essential that only one program at a time actually succeeds in entering a monitor procedure, and any subsequent call must be held up until the previous call has been completed."

It is, therefore, sometimes necessary to delay a program wishing to acquire a resource which is not available, and to resume that program after some other program has released the resource required. Thus monitors offer a "wait" and a "signal" operation. The "wait" operation causes the calling program to be delayed. The "signal" operation causes exactly one of the waiting programs to be resumed. If there are no waiting programs, the signal operation has no effect. In order to enable other programs to release resources during a wait operation, this operation must relinquish the mutual exclusion which would otherwise prevent entry to the releasing procedure.

In *Java*, the monitors are the objects. The monitor procedures, whose execution is mutually exclusive, can be declared by the modifier `synchronized`. Each object has a *lock* which can be owned by at most one *thread*, i.e., by at most one of the concurrently running processes. Synchronized methods of an object can be invoked only by a thread that owns the lock of that object. If the thread does not own the lock, it has to wait until the lock gets free. A thread owning the lock of an object can recursively invoke several synchronized methods of that object; this corresponds to the notion of reentrant monitors.

Besides mutual exclusion through the usage of the lock-mechanism for synchronized methods, *Java* objects offer the monitor methods wait, notify, and notifyAll. A thread owning the lock of an object can block itself ("go to sleep") and free the lock by invoking wait on the given object. The blocked thread can be reactivated by another thread owning the object's lock via the object's notify method, which corresponds to the "signal" operation of Hoare; the reactivated thread must reapply for the lock before it may continue its execution. The method notifyAll generalizes notify in that it notifies all threads blocked on the object.

It is often said that synchronized methods and the wait / notify constructs together implement Hoare's monitors. But there are some important differences between Hoare's monitors and *Java*'s monitors [Jok98].

Signaling in Hoare's monitor concept lets the signaled thread continue its execution *immediately* after the lock gets free, so if some thread is waiting for a resource it will obtain it. From [Hoa74] we quote: "We [...] need a 'wait' operation, issued from inside a procedure of the monitor, which causes the calling program to be delayed; and a 'signal' operation, also issued from inside a procedure of the same monitor, which causes exactly one of the waiting programs to be resumed immediately. [...] we decree that a signal operation be followed immediately by resumption of a waiting program, without possibility of an intervening procedure call from yet a third program. It is only in this way that a waiting program has an absolute guarantee that it can acquire the resource just released by the signaling program without any danger that a third program will interpose a monitor entry and seize the resource instead."

Java's notify method differs from the signal operation of Hoare. Quoting from [GJS96]: "The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object."

This difference implies that the following simple resource-allocation program[2] [Jok98, SG94] would assure mutual exclusion using Hoare's monitors,

[2]For readability, we omit the code for catching of InterruptedException for the call of the

but does not work correctly in *Java*:

```
public class Resource{
    private boolean busy = false;

    public synchronized void acquire(){
        if (busy){ wait(); }
        busy = true;
    }

    public synchronized void release(){
        busy = false;
        notify();
    }
}
```

Consider the following situation:

1. Initially busy is false.

2. Thread t_1 calls acquire and gets the ownership of the resource. The variable busy gets assigned the value true, and acquire returns.

3. Thread t_2 calls also acquire. Since busy is true, it goes to sleep.

4. Thread t_1 gets done with the resource and calls release. The variable busy gets assigned false, t_1 wakes up t_2, and release returns.

5. Now t_1 wants the resource again and calls acquire. The variable busy gets assigned true, and acquire returns.

6. After t_1 gives the lock free, t_2 may continue after the wait-statement. It will set busy to true again and returns.

Thus both t_1 and t_2 have access to the resource. If notify had guaranteed that t_2 is the next one running, as defined by Hoare, the program had worked correctly. The program can be made correct by replacing "if (busy)" by "while (busy)" so that the busy-state will be checked again every time the wait call returns.

Another main point in Hoare's article is a notion of a *condition variable*: A thread can be put to sleep waiting for a condition to happen. *Java* allows only the monitor itself as such a condition. As a consequence, in *Java* we cannot distinguish between threads waiting for different kinds of events in the same object. The following example makes the difference clear. It is a *Java* implementation of a simple producer-consumer example [Jok98, CW96]:

```
class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) { wait(); }
        available = false;
        notify();
        return contents;
    }
```

wait method.

```
    public synchronized void put(int value) {
        while (available == true) { wait(); }
        contents = value;
        available = true;
        notify();
    }
}
```

The following interleaving leads to a deadlock:

1. Initially, available is false.
2. Two consumer threads t_1^c and t_2^c both call get. Since available is false, both go to sleep.
3. A producer thread t^p calls put, sets available to true, awakes t_1^c and returns.
4. Now the producer t^p calls put again and, since available is true, goes to sleep.
5. The notified thread t_1^c continues, see that available is true, sets it to false, and notifies t_2^c.
6. The consumer t_1^c calls get again, available is false, so it goes to sleep.
7. Finally, the notified thread t_2^c continues its execution, sees that available is false, and goes to sleep again.

Now we have a deadlock, since all three threads are waiting for notification and there are no other threads left which could notify. What went wrong? After consuming, the consumer thread t_1^c wanted to awake the producer thread t^p, but because the producer was waiting in the same object as the consumer t_2^c, the notification reached t_2^c instead of t^p. With condition variables one could distinguish between different threads waiting for different events. In *Java* such condition variables are not available. The program can be corrected by replacing all notify by notifyAll. In this case, every sleeping thread will be awaken by every notification, and they all test their while-condition to make sure that one of them gets the correct wake-up call.

1.3 Hoare logic

Most of the research in program verification concentrates on the verification of *safety* properties [AS87] of programs. Such properties assert that the program never reaches some unexpected "bad" states. A typical safety property is deadlock freedom, asserting that the program cannot enter a deadlock state. Another example is mutual exclusion, expressing that two processes are never simultaneously in their critical sections.

The history of *assertional* proof methods for the verification of safety properties of programs goes back to Floyd [Flo67]. He introduced the concept of *partial correctness* for sequential programs. Each control point of the program is *annotated* with an assertion which should hold whenever control is at that point. Assertions attached to the control points in front of and after a statement are called the *pre-* and the *postcondition* of the statement. The program is partially correct, if for each *terminating* computation starting in a state satisfying the program's precondition, the final state satisfies the program's postcondition.

Hoare [Hoa69] recast Floyd's method into a logical framework. Whereas Floyd considered programs with an arbitrary control structure (flowcharts), Hoare's approach is based upon the structural decomposition of structured programs. A formula in *Hoare logic* has the form $\{\varphi\}P\{\psi\}$, and means that if the program P starts its execution in a state satisfying the precondition φ and if it terminates, then its final state satisfies the postcondition ψ. Inference rules reduce the proof of such a formula to the proofs of similar formulas for individual program statements.

Ashcroft [Ash75] extended the assertional reasoning of Floyd to parallel programs. As in Floyd's method, one assigns to each control point an assertion, which should hold whenever control is at that point. However, due to concurrency, control now can stay simultaneously at different control points. Thus the simple locality of Floyd's method is lost. For concurrent programs, the annotation is viewed as a single *invariant*, and one must prove that executing each statement leaves this invariant true. To capture synchronization, the assertions may mention the control state explicitly.

Owicki and Gries [OG76] and Lamport [Lam77] developed a generalization of Hoare's method to concurrent programs with shared-variable concurrency. Additionally to Hoare's rules, they introduced a new rule for the parallel composition, which requires the composed programs to be *interference free*. Interference freedom for the parallel composition of n programs P_i, $i = 1, \ldots, n$, requires that the execution of any statement in any P_i with its precondition true leaves each assertion in the annotation of each P_j with $j \neq i$ true. The Owicki-Gries method avoids mentioning the control state in the annotation by introducing *auxiliary variables* to capture the control information. This leads to program *augmentation* which extends the program by assignments to auxiliary variables. A typical auxiliary variable is a "program counter" storing the current control point of execution. Its value gets updated in each computation step by additional *auxiliary assignments*.

The Owicki-Gries method has been adapted and extended by several research groups. For example, CSP (Communicating Sequential Processes) [Hoa78] was treated independently by Apt, Francez, and de Roever [AFdR80] and by Levin and Gries [LG81]. CSP allows synchronous communication between concurrent processes. Synchronous communication requires different proof-theoretical treatment as shared variable concurrency. The *cooperation test* collects conditions which assure the invariance of the properties of synchronously communicating processes.

Following Owicki and Gries, the general method to show correctness of a Hoare formula for concurrent programs is to find an *invariant* program property I such that:

1. the precondition implies I,

2. if the program starts in a state satisfying I, then every reachable state satisfies I, and

3. I implies the postcondition.

The first and last conditions are static, i.e., they depend only on the program syntax (or more exactly, on the annotation definition), whereas the second criterion is a dynamic property, which describes the run-time behavior of the program. The second point, the invariance of I, can be proven by induction showing that each *atomic* action executed in a state satisfying I terminates in a state in which I is true again.

This thesis combines and extends the results of [OG76], [Lam77], [AFdR80], and [LG81]. We formulate a proof system for a concurrent class-based object-oriented language —a *Java* sublanguage— allowing both shared-variable concurrency and synchronous communication in the form of method calls, as well as reentrant monitor synchronization and dynamic object and process creation.

In our class-based setting, the proof method requires an annotation of classes and their methods using a local assertion language (see Section 1.1), adhering to the principle of data encapsulation. A global invariant formulated in the global assertion language combines properties of objects, describing their communication structure.

The Hoare rules of our proof system are grouped into four groups:

1. *initial* correctness,

2. *local* correctness,

3. the *interference freedom test,* and

4. the *cooperation test.*

Initial correctness states that the program starts in a state which satisfies the program's precondition. Local correctness, as in Hoare's method, assures invariance of properties of a single process under its *own* execution.

The notion of interference freedom is introduced by Owicki and Gries and covers the effect of shared-variable concurrency. It assures that properties of processes are invariant under the execution of *other* concurrently running processes. Our interference freedom test extends that of the Owicki-Gries method to cover shared-variable concurrency in a class-based object-oriented setting allowing recursion and reentrant monitor synchronization.

Finally, the cooperation test deals with invariance of properties of *communicating* processes. Such rules for communication were introduced for CSP in [AFdR80] and in [LG81]. In our object-oriented language, communication via method call can take place between different objects. We model the rendezvous of a method call as two CSP-like communication between objects: The first communication invokes the method and passes on the actual parameter values to the callee, whereas the second communication returns the control and the result of the method, i.e., the return value, from the invoked method to the caller. Note that while in CSP communication takes place between *processes*, communication via method call is between *objects*. Processes, i.e., threads, communicate only via shared variables.

All verification condition groups together imply invariance of the whole program annotation under program execution.

1.4 Overview

The thesis is organized as follows: Chapter 2 describes syntax and semantics of a sequential sublanguage of *Java*. After introducing the assertional logic, we present a proof system for the sequential case. Chapter 3 extends the results to a concurrent sublanguage. The language introduced in Chapter 4 includes *Java*'s monitor-synchronization mechanism. The verification conditions in the above sections are formulated as standard Hoare triples. Section 5 reformulates the verification conditions to logical implications using a weakest-precondition calculus. Soundness and relative completeness are discussed in Chapter 6. Chapter 7 shows how we can prove deadlock freedom, and gives some examples. Chapter 8 describes possible extensions of the proof system to cover additional language features of *Java*. We introduce the verification tool and sketch its use in Chapter 9. Section 10 contains some concluding remarks. The appendix contains proofs of those theorems which state soundness and relative completeness of the proof system for multithreaded *Java* programs with monitor synchronization.

Chapter 2

The sequential language

In this chapter we introduce a sequential sublanguage *Java*$_{seq}$ of *Java*. The language allows assignments, dynamic object creation, aliasing, method invocation, and recursion. We define the syntax in Section 2.1, and the semantics in Section 2.2. After defining the assertion language in Section 2.3, we introduce a proof system for verifying safety properties of programs written in the language in Section 2.4. Section 2.5, finally, concludes with some remarks and related work.

Programs, as in *Java*, are given by a collection of classes containing instance variable and method declarations. *Instances* of the classes, i.e., *objects,* are dynamically created, and communicate via *method invocation,* i.e., synchronous message passing.

We ignore in *Java*$_{seq}$ the issues of *concurrency*, *inheritance*, and consequently subtyping, overriding, and late-binding. For simplicity, we neither allow method *overloading*, i.e., we require that each method name has been assigned a unique list of formal parameter types and a return type. In short, being concerned with the verification of the run-time behavior, we assume a simple *monomorphic* type discipline for *Java*$_{seq}$.

2.1 Syntax

Java$_{seq}$ is a strongly typed language; besides class types c, it supports booleans Bool and integers Int as primitive types, and pairs $t \times t$ and lists $\mathsf{list}\, t$ as composed types. We use the type Void for methods without return value. Since *Java*$_{seq}$ is strongly typed, all program constructs of the abstract syntax are silently assumed to be well-typed. In other words, we work with a type-annotated abstract syntax where we omit the explicit mentioning of types when this causes no confusion.

For each type, the corresponding value domain is equipped with a standard set of operators with typical element f. Each operator f has a unique type $t_1 \times \cdots \times t_n \to t$ and a fixed interpretation f, where constants are operators of zero arity. Apart from the standard repertoire of arithmetical and boolean operations, the set of operators also contains operations on tuples and sequences like projection, concatenation, etc.

For variables, we notationally distinguish between *instance variables* and *local (temporary) variables*. Instance variables hold the state of an object and exist throughout the object's lifetime. Local variables are stack-allocated; they play the role of formal parameters and variables of method definitions and only exist during the execution of the method to which they belong. We define $IVar$ to be the set of instance variables with typical element x, and $TVar$ as the set of local variables with typical elements $u, u', v, \ldots$. Let $Var = IVar \dot{\cup} TVar$ with typical element y be the set of program variables, where $\dot{\cup}$ is the disjoint union operator.

The abstract syntax is summarized in Table 2.1. It slightly differs from the corresponding *Java* syntax. Though we use the abstract syntax for the theoretical part of this work, our tool supports *Java* syntax (cf. Chapter 9).

$$
\begin{array}{rcl}
e & ::= & x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(e, \ldots, e) \\
e_{ret} & ::= & \epsilon \mid e \\
stm & ::= & x := e \mid u := e \mid u := \mathsf{new}^c \\
 & \mid & u := e.m(e, \ldots, e) \mid e.m(e, \ldots, e) \\
 & \mid & \epsilon \mid stm; stm \mid \mathsf{if}\ e\ \mathsf{then}\ stm\ \mathsf{else}\ stm\ \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ stm\ \mathsf{od} \ldots \\
meth & ::= & m(u, \ldots, u)\{\ stm; \mathsf{return}\ e_{ret}\} \\
meth_{\mathsf{run}} & ::= & \mathsf{run}()\{\ stm; \mathsf{return}\ \} \\
class & ::= & \mathsf{class}\ c\{meth \ldots meth\} \\
class_{\mathsf{main}} & ::= & \mathsf{class}\ c\{meth \ldots meth\ meth_{\mathsf{run}}\} \\
prog & ::= & class \ldots class\ class_{\mathsf{main}}
\end{array}
$$

Table 2.1: $Java_{seq}$ abstract syntax

Besides using instance and local variables, *expressions* $e \in Exp$ are built from the self-reference this, the empty reference null, and from subexpressions using the given operators. To support a clean interface between internal and external object behavior, $Java_{seq}$ does not allow qualified references to instance variables (cf. Section 1.1). Note that all expressions of the language are side-effect free, i.e., their evaluation does not modify the program state. Only the execution of statements may have such an effect.

As *statements* $stm \in Stm$, we allow assignments, object creation, method invocation, and standard control constructs like sequential composition, conditional statements, and iteration. We write ϵ for the empty statement.

A *method* definition $m(u_1, \ldots, u_n)\{stm; \mathsf{return}\ e_{ret}\}$ consists of a method name m, a list of formal parameters $u_1, \ldots, u_n$, and a method body of the form $stm; \mathsf{return}\ e_{ret}$, i.e., we require that method bodies are terminated by a single return statement of the form return or return e, giving back the control and possibly a return value. We sometimes syntactically omit return statements without return value in method definitions. The set $Meth_c$ contains the methods of class c. We denote the body of method m of class c by $body_{m,c}$. We sometimes explicitly mention the types of return value and formal parameters in *Java*-style $t\ m(t_1\ u_1, \ldots, t_n\ u_n)$.

A *class* is defined by its name c and its methods, whose names are assumed to be distinct. A *program,* finally, is a collection of class definitions having different class names, where a main class $class_{\mathsf{main}}$ defines by its run method the entry point of the program execution. We call the body of the run method of the main class the *main statement* of the program.[1] The run method cannot be called.

The set $IVar_c$ of instance variables of a class c is given implicitly by the instance variables occurring in the class; the set of local variables of method declarations is given similarly. In the examples we sometimes explicitly define the instance and local variables in *Java*-style: the declaration $t\ y$; in classes outside of method definitions declare y as an instance variable of type t of the class, whereas the same declaration inside of a method specifies y as a local variable.

Besides the mentioned simplifications of the type system, we impose for technical reasons the following restrictions: We require that method invocation statements contain only local variables, i.e., that none of the expressions $e_0, \ldots, e_n$ in a method invocation $e_0.m(e_1, \ldots, e_n)$ contains instance variables. Furthermore, formal parameters must not occur on the left-hand side of assignments. These restrictions imply that during the execution of a method the values of the actual and formal parameters are not changed. Finally, the result of object creation and method invocation may not be assigned to instance variables. This restriction allows a proof system with separated verification conditions for interference freedom and cooperation. The above restrictions could be relaxed, without loosing the mentioned modularity, but it would increase the complexity of the proof system (see Section 8.2).

It should be clear that it is possible to transform a program to adhere to the above restrictions at the expense of additional local variables and thus new interleaving points. To demonstrate such a transformation, assume the following class:

```
class C{
    Int x1;

    Void m1(C o){
        x1 := o.m2(x1);
        return
    }

    Int m2(Int u){
        return u+1
    }
```

[1] In *Java*, the entry point of a program is given by the static main method of the main class. Relating the abstract syntax to that of *Java*, we assume that the main class is a Thread-class whose main method just creates an instance of the main class and starts its thread. The reason to make this restriction is, that *Java*'s main method is static, but our proof system does not support static methods and variables.

```
}
```

The following transformation satisfies the requirements, but inserts additional control points before and after the call in method `m1`:

```
class C{
    Int x1;

    Void m1(C o){
        Int u,v;

        u := x1;
        v := o.m2(u);
        x1 := v;
        return
    }

    Int m2(Int u){
        return u+1
    }
}
```

2.2 Semantics

There are several ways to describe the semantics of programs formally. Three commonly used approaches are operational, denotational, and axiomatic semantics.

An *operational* semantics defines the meaning of a program by a set of rules specifying how the program state changes while executing a program. The overall state is typically divided into a number of components, e.g. stack, heap, registers etc. Each rule specifies certain preconditions on the contents of some components and their new contents after the application of the rule. One of the earliest papers was by McCarthy [McC65]. Operational semantics is quite concrete with a low-level description of program execution. A *structural* approach to operational semantics was initiated by Plotkin [Plo81].

Denotational semantics is a technique for describing the meaning of programs in terms of mathematical functions on programs and program components. Programs are translated into functions about which properties can be proved using the standard mathematical theory of functions, and especially domain theory. Landin [Lan64, Lan65, Lan66] made major early steps towards denotational semantics. Some semantical problems appearing in connection with recursion were analyzed by Scott and lead to domain theory [Sco70, Sco76]; see also [Sto77].

The axiomatic semantics [Hoa69] defines the language semantics by a system of logical axioms and inference rules. In an axiomatic semantics not the meaning of a program but its properties are defined. Such a semantics directly aims to support program verification.

In this section, we define the *operational semantics* of Java$_{seq}$. After introducing the semantic domains, we describe states and configurations in the following section. The operational semantics is presented in Section 2.2.2. The meaning of a program is defined by a set of transition rules specifying how the program configuration changes while executing program statements.

2.2.1 States and configurations

Let Val^t be disjoint domains for the various types t. For class names c, the disjoint sets Val^c with typical elements $\alpha, \beta, \ldots$ denote infinite sets of *object identities.* The value of null in type c is $null^c \notin Val^c$. In general we just write $null$, when c is clear from the context. We define Val^c_{null} as $Val^c \dot\cup \{null^c\}$, and correspondingly for composed types. The set of all possible non-null values $\bigcup_t Val^t$ is written as Val, and Val_{null} denotes $\bigcup_t Val^t_{null}$. Let $Init : Var \rightarrow Val_{null}$ be a function assigning an initial value to each variable $y \in Var$, i.e., $null$, $false$, and 0, for class, boolean, and integer types, respectively, and analogously for composed types, where sequences are initially empty. We define $\mathsf{this} \notin Var$, such that the self-reference is not in the domain of $Init$.

The configuration of a program consists of the set of existing objects together with the values of their instance variables, and the configuration of the executing thread. Before formalizing the global configurations of a program, we define local states and local configurations. In the sequel we identify the occurrence of a statement in a program with the statement itself.

A *local state* $\tau \in \Sigma_{loc}$ of a method execution holds the values of the method's local variables and is modeled as a partial function of type $TVar \rightharpoonup Val_{null}$. We refer to local states of method m of class c by $\tau^{m,c}$. The initial local state $\tau^{m,c}_{init}$ assigns to each local variable u from its domain the value $Init(u)$. A *local configuration* (α, τ, stm) of a method of an object $\alpha \neq null$ specifies, in addition to its local state τ, its point of execution represented by the statement stm. A *thread configuration* ξ is a stack of local configurations $(\alpha_0, \tau_0, stm_0)(\alpha_1, \tau_1, stm_1) \ldots (\alpha_n, \tau_n, stm_n)$, representing the chain of method invocations of the given thread, i.e., process. We write $\xi \circ (\alpha, \tau, stm)$ for pushing a new local configuration onto the stack.

Objects are characterized by their *instance states* $\sigma_{inst} \in \Sigma_{inst}$ of type $IVar \dot\cup \{\mathsf{this}\} \rightharpoonup Val_{null}$ such that this is in the domain $dom(\sigma_{inst})$ of σ_{inst}. We write σ^c_{inst} to denote states of instances of class c. The semantics will maintain $\sigma^c_{inst}(\mathsf{this}) \in Val^c$ as invariant. The initial instance state $\sigma^{c,init}_{inst}$ assigns a value from Val^c to this, and to each of its remaining instance variables x the value $Init(x)$.

A *global state* $\sigma \in \Sigma$ of type $(\bigcup_c Val^c) \rightharpoonup \Sigma_{inst}$ stores for each currently *existing* object, i.e., an object belonging to the domain of σ, its instance state. The set of existing objects of type c in a state σ is given by $Val^c(\sigma)$, and $Val^c_{null}(\sigma) = Val^c(\sigma) \dot\cup \{null^c\}$. For the remaining types, $Val^t(\sigma)$ and $Val^t_{null}(\sigma)$ are defined correspondingly. We refer to the set $\bigcup_t Val^t(\sigma)$ by $Val(\sigma)$; $Val_{null}(\sigma)$ denotes $\bigcup_t Val^t_{null}(\sigma)$. The instance state of an object $\alpha \in Val(\sigma)$ is given by $\sigma(\alpha)$ with the invariant property $\sigma(\alpha)(\mathsf{this}) = \alpha$. We require that, given a global state, no instance variable in any of the existing objects refers to a non-existing object, i.e., $\sigma(\alpha)(x) \in Val_{null}(\sigma)$ for all classes c and objects $\alpha \in Val^c(\sigma)$. This will be an invariant of the operational semantics of the next section.

A *global configuration* $\langle T, \sigma \rangle$ describes the currently existing objects by the global state σ, where the set T contains the configuration of the executing thread. For the concurrent languages of the later sections, T will be the set of configurations of all currently executing threads. Analogously to the restriction on global states, we require that local configurations (α, τ, stm) in $\langle T, \sigma\rangle$ refer only to existing object identities, i.e., $\alpha \in Val(\sigma)$ and $\tau(u) \in Val_{null}(\sigma)$ for all variables u from the domain of τ; again this will be an invariant of the operational semantics. In the following, we write $(\alpha, \tau, stm) \in T$ if there exists a local configuration (α, τ, stm) within one of the execution stacks of T.

The semantic function $[\![\_]\!]_{\mathcal{E}}^{\_,\_} : (\Sigma_{inst} \times \Sigma_{loc}) \to (Exp \rightharpoonup Val_{null})$ evaluates in the context of an *instance local* state (σ_{inst}, τ) expressions containing variables from $dom(\sigma_{inst}) \cup dom(\tau)$, where $dom(f)$ denotes the domain of the function f. Instance variables x and local variables u are evaluated to $\sigma_{inst}(x)$ and $\tau(u)$, respectively, this evaluates to $\sigma_{inst}(\mathsf{this})$, and null has the *null*-reference as value, where composed expressions are evaluated by homomorphic lifting (see Table 2.2).

$$\begin{array}{rcl}
[\![x]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} & = & \sigma_{inst}(x) \\
[\![u]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} & = & \tau(u) \\
[\![\mathsf{this}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} & = & \sigma_{inst}(\mathsf{this}) \\
[\![\mathsf{null}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} & = & null \\
[\![\mathsf{f}(e_1, \ldots, e_n)]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} & = & f([\![e_1]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}, \ldots, [\![e_n]\!]_{\mathcal{E}}^{\sigma_{inst},\tau})
\end{array}$$

Table 2.2: Semantics of program expressions

We denote by $\tau[u \mapsto v]$ the local state which assigns the value v to u and agrees with τ on the values of all other variables; $\sigma_{inst}[x \mapsto v]$ is defined analogously, where $\sigma[\alpha.x \mapsto v]$ results from σ by assigning v to the instance variable x of object $\alpha \in Val(\sigma)$. We use these operators analogously for vectors of variables. We use $\tau[\vec{y} \mapsto \vec{v}]$ also for arbitrary variable sequences, where instance variables are untouched; $\sigma_{inst}[\vec{y} \mapsto \vec{v}]$ and $\sigma[\alpha.\vec{y} \mapsto \vec{v}]$ are defined analogously. Finally, for global states, $\sigma[\alpha \mapsto \sigma_{inst}]$ equals σ except on α; in case $\alpha \notin Val(\sigma)$, the operation extends the set of existing objects by α, which has its instance state initialized to σ_{inst}.

2.2.2 Operational semantics

Before having a closer look at the semantical rules for the transition relation $\longrightarrow$, let us start by defining the entry point of a program. The initial configuration $\langle T_0, \sigma_0\rangle$ of a program satisfies $dom(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma_{inst}^{c,init}[\mathsf{this} \mapsto \alpha]$, and $T_0 = \{(\alpha, \tau_{init}^{\mathsf{run},c}, body_{\mathsf{run},c})\}$, where c is the main class, and $\alpha \in Val^c$ is the initial object.

We call a configuration $\langle T, \sigma\rangle$ of a program *reachable* if there exists a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle$ such that $\langle T_0, \sigma_0\rangle$ is the initial configuration of the program and $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$. A local configuration

$(\alpha, \tau, stm) \in T$ is *enabled* in $\langle T, \sigma\rangle$, if it can be executed, i.e., if there is a computation step $\langle T, \sigma\rangle \rightarrow \langle T', \sigma'\rangle$ executing stm in the local state τ and object α.

The operational semantics of $Java_{seq}$ is given inductively by the rules of Table 2.3 as transitions between global configurations. The rules are formulated in such a way that we can reuse them also for the concurrent languages of the later sections. Note that for the sequential language, the sets T in the rules are empty, since there is only one single thread in global configurations. We omit the rules for the remaining sequential constructs —sequential composition, conditional statement, and iteration— since they are standard. Remember that $\dot\cup$ is the disjoint union operator.

$$\frac{}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, x{:=}e; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm)\}, \sigma[\alpha.x \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]\rangle} \ \textsc{Ass}_{inst}$$

$$\frac{}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, u{:=}e; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau[u \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}], stm)\}, \sigma\rangle} \ \textsc{Ass}_{loc}$$

$$\frac{\beta \in Val^c \setminus Val(\sigma) \qquad \sigma_{inst} = \sigma_{inst}^{c,init}[\mathsf{this} \mapsto \beta] \qquad \sigma' = \sigma[\beta \mapsto \sigma_{inst}]}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, u{:=}\mathsf{new}^c; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau[u \mapsto \beta], stm)\}, \sigma'\rangle} \ \textsc{New}$$

$$\frac{\begin{array}{c} m(\vec{u})\{\ body\ \} \in Meth_c \\ \beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in Val^c(\sigma) \qquad \tau' = \tau_{init}^{m,c}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \end{array}}{\begin{array}{l} \langle T \dot\cup \{\xi \circ (\alpha, \tau, u := e_0.m(\vec{e}); stm)\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{receive}\, u; stm) \circ (\beta, \tau', body)\}, \sigma\rangle \end{array}} \ \textsc{Call}$$

$$\frac{\tau'' = \tau[u_{ret} \mapsto [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma(\beta),\tau'}]}{\begin{array}{l} \langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{receive}\, u_{ret}; stm) \circ (\beta, \tau', \mathsf{return}\ e_{ret})\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau'', stm)\}, \sigma\rangle \end{array}} \ \textsc{Return}$$

$$\frac{}{\langle T \dot\cup \{(\alpha, \tau, \mathsf{return})\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{(\alpha, \tau, \epsilon)\}, \sigma\rangle} \ \textsc{Return}_{run}$$

Table 2.3: $Java_{seq}$ operational semantics

Assignments to instance or local variables update the corresponding state component, i.e., either the instance state or the local state (rules $\textsc{Ass}_{inst}$ and $\textsc{Ass}_{loc}$). Object creation by $u := \mathsf{new}^c$, as shown in rule New, creates a new object of type c with a fresh identity stored in the local variable u, and initializes the instance variables of the new object. Invoking a method extends the call chain by a new local configuration (rule Call). We use the auxiliary statement $\mathsf{receive}\, u$ to remember the variable in which the result of the invoked method will be stored at returning. After initializing the local state and passing the parameters, the thread begins to execute the method body. When returning from a method call (rule Return), the callee evaluates its return expression and passes it to the caller, which subsequently updates its local state. The method body terminates its execution and the caller can continue. We have similar rules not shown in the table for the invocation of methods without return value. The executing thread ends its lifespan by returning from the run method of the initial object (rule $\textsc{Return}_{run}$).

2.3 The assertion language

In this section we introduce *assertions* to specify program properties. The assertion logic consists of a *local* and a *global* sublanguage. *Local* assertions describe instance local states, and are used to annotate methods in terms of their local variables and of the instance variables of the class to which they belong. *Global* assertions describe the global state, i.e., a whole system of objects and their communication structure.

In the assertion language we add the type Object as the supertype of all classes. Note that we allow this type solely in the assertion language, but not in the programming language, thus preserving the assumption of monomorphism.

2.3.1 Syntax

In the language of assertions, we introduce a countably infinite set $LVar$ of well-typed *logical variables* with typical element z, where we assume that instance variables, local variables, and this are not in $LVar$. We use $LVar^t$ for the set of logical variables of type t. Logical variables are used for quantification in both the local and the global language. Besides that, they are used as free variables to represent local variables in the global assertion language: To express a local property on the global level, each local variable in a given local assertion will be replaced by a fresh logical variable.

Table 2.4 defines the syntax of the assertion language. For readability, we use the standard syntax of first-order logic in the theoretical part; the *Verger* tool supports an adaptation of *JML* (cf. Chapter 9).

Local expressions $e \in LExp$ are expressions of the programming language possibly containing logical variables. The set of local expressions of type t is denoted by $LExp^t$. Abusing our notation, we use $e, e', \ldots$ not only for program expressions of Table 2.1, but also for typical elements of local expressions. *Local assertions* $p, p', q, \ldots \in LAss$ are standard logical formulas over boolean local expressions. We allow three forms of quantification over logical variables: Unrestricted quantification $\exists z.\ p$ is solely allowed for domains without object references, i.e., the type of z is required to be Int, Bool, or a composed type built from them. For reference types c, this form of quantification is not allowed, as for those types the existence of a value dynamically depends on the *global* state, something one cannot speak about on the local level, or more formally: Disallowing unrestricted quantification for object types ensures that the value of a local assertion indeed only depends on the values of the instance and local variables, but not on the global state. Nevertheless, one can assert the existence of objects on the local level satisfying a predicate, provided one is explicit about the set of objects to range over. Thus, the restricted quantifications $\exists z \in e.\ p$ and $\exists z \sqsubseteq e.\ p$ assert the existence of an element, respectively, the existence of a subsequence of a given sequence e, for which a property p holds.

Global expressions $E, E', \ldots \in \mathit{GExp}$ are constructed from logical variables, null, operator expressions, and qualified references $E.x$ to instance variables x of objects E. We write GExp^t for the set of global expressions of type t. *Global assertions* $P, Q, \ldots \in \mathit{GAss}$ are logical formulas over boolean global expressions. Unlike the local language, the meaning of the global one is defined in the context of a global state. Thus unrestricted quantification is allowed for all types and is interpreted to range over the set of *existing* values and *null*, i.e., the set of values $\mathit{Val}_{null}(\sigma)$ in a global configuration $\langle T, \sigma \rangle$.

$$
\begin{array}{rcll}
e & ::= & z \mid x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(e, \ldots, e) & e \in \mathit{LExp} \\
p & ::= & e \mid \neg p \mid p \wedge p & \\
 & \mid & \exists z.\ p \mid \exists z \in e.\ p \mid \exists z \sqsubseteq e.\ p & p \in \mathit{LAss} \\
 & & & \\
E & ::= & z \mid \mathsf{null} \mid \mathsf{f}(E, \ldots, E) \mid E.x & E \in \mathit{GExp} \\
P & ::= & E \mid \neg P \mid P \wedge P \mid \exists z.\ P & P \in \mathit{GAss}
\end{array}
$$

Table 2.4: Syntax of assertions

We sometimes write quantification over t-typed values in the form $\exists (z : t).\, p$ to make the domain of the quantification explicit; we use the same notation also in the global language. We use $\forall z.\, p$ for $\neg\exists z.\, \neg p$.

2.3.2 Semantics

Next, we define the interpretation of the assertion language. The semantics is fairly standard, except that we have to cater for dynamic object creation when interpreting quantification.

Logical variables are interpreted relative to a *logical environment* $\omega \in \Omega$, that is, a partial function of type $\mathit{LVar} \rightharpoonup \mathit{Val}_{null}$, assigning values to logical variables. We denote by $\omega[\vec{z} \mapsto \vec{v}]$ the logical environment that assigns the values $\vec{v}$ to the logical variables $\vec{z}$, and agrees with ω on all other variables. Similarly as for local and instance state updates, we use also $\omega[\vec{y} \mapsto \vec{v}]$ for arbitrary variable sequences $\vec{y}$ to denote the logical environment which assigns to each logical variable in $\vec{y}$ the corresponding value in $\vec{v}$, and agrees with ω on all other variable values. For a logical environment ω and a global state σ we say that ω refers only to values existing in σ, if $\omega(z) \in Val_{null}(\sigma)$ for all $z \in dom(\omega)$. This property matches with the definition of quantification which ranges only over existing values and *null*, and with the fact that in reachable configurations local variables may refer only to existing values or to *null*.

The semantic function $[\![\_]\!]_{\mathcal{L}}^{\_,\_,\_}$ of type $(\Omega \times \Sigma_{inst} \times \Sigma_{loc}) \to (LExp \cup LAss \rightharpoonup Val_{null})$ evaluates local expressions and assertions in the context of a logical environment ω and an instance local state (σ_{inst}, τ) (cf. Table 2.5). The evaluation function is defined for expressions and assertions that contain only variables from $dom(\omega) \cup dom(\sigma_{inst}) \cup dom(\tau)$. The instance local state provides the context for giving meaning to programming language expressions as defined by the semantic function $[\![\_]\!]_{\mathcal{E}}^{\_,\_}$; the logical environment evaluates logical variables. An unrestricted quantification $\exists z.\, p$ with $z \in LVar^t$ evaluates to *true* in the logical environment ω and instance local state (σ_{inst}, τ) iff there exists a value $v \in Val^t$ such that p holds in the logical environment $\omega[z \mapsto v]$ and instance local state (σ_{inst}, τ), where for the type t of z only Int, Bool, or composed types built from them are allowed. The evaluation of a restricted quantification $\exists z \in e.\, p$ with $z \in LVar^t$ and $e \in LExp^{\mathsf{list}\,t}$ is defined analogously, where the existence of an element in the sequence is required. An assertion $\exists z \sqsubseteq e.\, p$ with $z \in LVar^{\mathsf{list}\,t}$ and $e \in LExp^{\mathsf{list}\,t}$ states the existence of a subsequence of e for which p holds. In the following we also write $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ for $[\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true$. By $\models_{\mathcal{L}} p$ we express that $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ holds for arbitrary logical environments, instance states, and local states.

$$
\begin{array}{rcl}
[\![z]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \omega(z) \\
[\![x]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \sigma_{inst}(x) \\
[\![u]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \tau(u) \\
[\![\mathsf{this}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \sigma_{inst}(\mathsf{this}) \\
[\![\mathsf{null}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & null \\
[\![\mathsf{f}(e_1,\ldots,e_n)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & f([\![e_1]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau},\ldots,[\![e_n]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}) \\
([\![\neg p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}=true) & \text{iff} & ([\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}=false) \\
([\![p_1 \wedge p_2]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}=true) & \text{iff} & ([\![p_1]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}=true \text{ and } [\![p_2]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}=true) \\
([\![\exists z.\, p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}=true) & \text{iff} & ([\![p]\!]_{\mathcal{L}}^{\omega[z\mapsto v],\sigma_{inst},\tau}=true \text{ for some } v \in Val_{null}) \\
([\![\exists z{\in}e.\, p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}=true) & \text{iff} & ([\![z{\in}e{\wedge}p]\!]_{\mathcal{L}}^{\omega[z\mapsto v],\sigma_{inst},\tau}=true \text{ for some } v{\in} Val_{null}) \\
([\![\exists z{\sqsubseteq}e.\, p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}=true) & \text{iff} & ([\![z{\sqsubseteq}e{\wedge}p]\!]_{\mathcal{L}}^{\omega[z\mapsto v],\sigma_{inst},\tau}=true \text{ for some } v{\in} Val_{null})
\end{array}
$$

Table 2.5: Local evaluation

Since *global* assertions do not contain local variables and non-qualified references to instance variables, the global assertional semantics does not refer to instance local states but to global states. The semantic function $[\![\_]\!]_{\mathcal{G}}^{-,-}$ of type $(\Omega \times \Sigma) \rightharpoonup (GExp \cup GAss \rightharpoonup Val_{null})$, shown in Table 2.6, gives meaning to global expressions and assertions in the context of a logical environment ω and a global state σ. To be well-defined, ω is required to refer only to values existing in σ, and the expression respectively assertion may only contain free variables from the domain of ω. Logical variables, null, and operator expressions are evaluated analogously to local assertions. The value of a global expression $E.x$ is given by the value of the instance variable x of the object referred to by the expression E. The evaluation of an expression $E.x$ is defined only if E refers to an object existing in σ. Note that when E and E' refer to the same object, that is, E and E' are *aliases*, then $E.x$ and $E'.x$ denote the same variable. The semantics of negation and conjunction is standard. A quantification $\exists z.\ P$ with $z \in LVar^t$ evaluates to true in the context of ω and σ if P evaluates to true in the context of $\omega[z \mapsto v]$ and σ, for some value $v \in Val^t_{null}(\sigma)$. Note that quantification over objects ranges over the set of *existing* objects and *null*, only.

$$
\begin{array}{rcl}
[\![z]\!]_{\mathcal{G}}^{\omega,\sigma} & = & \omega(z) \\
[\![\mathsf{null}]\!]_{\mathcal{G}}^{\omega,\sigma} & = & null \\
[\![\mathsf{f}(E_1,\ldots,E_n)]\!]_{\mathcal{G}}^{\omega,\sigma} & = & f([\![E_1]\!]_{\mathcal{G}}^{\omega,\sigma},\ldots,[\![E_n]\!]_{\mathcal{G}}^{\omega,\sigma}) \\
[\![E.x]\!]_{\mathcal{G}}^{\omega,\sigma} & = & \sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) \\
([\![\neg P]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P]\!]_{\mathcal{G}}^{\omega,\sigma} = false) \\
([\![P_1 \wedge P_2]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P_1]\!]_{\mathcal{G}}^{\omega,\sigma} = true \text{ and } [\![P_2]\!]_{\mathcal{G}}^{\omega,\sigma} = true) \\
([\![\exists z.\ P]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P]\!]_{\mathcal{G}}^{\omega[z \mapsto v],\sigma} = true \text{ for some } v \in Val_{null}(\sigma))
\end{array}
$$

Table 2.6: Global evaluation

For a global state σ and a logical environment ω referring only to values existing in σ we write $\omega, \sigma \models_{\mathcal{G}} P$ when P is true in the context of ω and σ. We write $\models_{\mathcal{G}} P$ if P holds for arbitrary global states σ and logical environments ω referring only to values existing in σ.

To express a local property p in the global assertion language, we define the lifting substitution $p[z/\mathsf{this}]$ by simultaneously replacing in p all occurrences of the self-reference this by the logical variable z, which is assumed not to occur in p, and transforming all occurrences of instance variables x into qualified references $z.x$. For notational convenience we view the local variables occurring in the global assertion $p[z/\mathsf{this}]$ as logical variables. Formally, these local variables are replaced by fresh logical variables. For unrestricted quantifications $(\exists z'.\ p)[z/\mathsf{this}]$ the substitution applies to the assertion p. Local restricted quantifications are transformed into global unrestricted ones where the relations $\in$ and $\sqsubseteq$ are expressed at the global level as operators. The main cases of the substitution are defined as follows:

$$\begin{array}{rcl}
\mathsf{this}[z/\mathsf{this}] & = & z \\
x[z/\mathsf{this}] & = & z.x \\
u[z/\mathsf{this}] & = & u \\
(\exists z'.\ p)[z/\mathsf{this}] & = & \exists z'.\ p[z/\mathsf{this}] \\
(\exists z' \in e.\ p)[z/\mathsf{this}] & = & \exists z'.\ (z' \in e[z/\mathsf{this}] \wedge p[z/\mathsf{this}]) \\
(\exists z' \sqsubseteq e.\ p)[z/\mathsf{this}] & = & \exists z'.\ (z' \sqsubseteq e[z/\mathsf{this}] \wedge p[z/\mathsf{this}])\ ,
\end{array}$$

where z is fresh. We write $P(z)$ for $p[z/\mathsf{this}]$, and similarly for expressions.

This substitution will be used to combine properties of instance local states on the global level. The substitution preserves the meaning of local assertions, provided the meaning of the local variables is matchingly represented by the logical environment:

Lemma 2.3.1 (Lifting substitution) *Let σ be a global state, ω and τ a logical environment and local state, both referring only to values existing in σ. Let furthermore p be a local assertion containing local variables $\vec{u}$. If $\tau(\vec{u}) = \omega(\vec{u})$ and z a fresh logical variable, then*

$$\omega, \sigma \models_{\mathcal{G}} p[z/\mathsf{this}] \qquad \textit{iff} \qquad \omega, \sigma(\omega(z)), \tau \models_{\mathcal{L}} p\ .$$

The proof can be found in Appendix A.1.

2.4 The proof system

Program verification is concerned with proving that a particular program meets its specification. In this section we develop a deductive Hoare-style proof system for the sequential language $Java_{seq}$. A general introduction to Hoare-style verification can be found in Section 1.3.

The proof of correctness of a program property consists of three steps. First, the required property must be specified by augmenting and annotating the program, i.e., by extending the program with auxiliary assignments which do not influence the control flow of the original program, and by attaching predicates to syntactical program constructs. An augmented and annotated program is called a proof outline. Second, the proof system must be applied to the particular proof outline, resulting in a set of verification conditions. Finally, the verification conditions must be proven.

In this section we introduce the proof system; its application and tool support are discussed in Chapter 9. The proof system has to accommodate dynamic object creation, aliasing, method invocation, and recursion. The following section defines how to augment and annotate programs resulting in proof outlines; Section 2.4.2 describes the proof method.

For technical convenience, we first formulate verification conditions as standard Hoare triples of the form $\{p\}\ stm\ \{q\}$, where the statement stm is a multiple assignment or the sequential composition of multiple assignments, representing state updates. In verification conditions formulated in the local assertion language, the multiple assignments in the Hoare triples may refer to instance and local variables. The statements in global conditions may use logical variables and qualified references to instance variables. Remember that local variables are represented in the global language by logical variables.

Example 2.4.1 *The Hoare triple* $\{u > 0 \wedge v > 0\}\ x := u * v\ \{x > 0\}$*, formulated in the local language, states that if both u and v have positive values, then after the execution of the assignment* $x := u * v$ *the value of* x *is positive.*

The Hoare triple $\{u > 0 \wedge v > 0\}\ z.x := u * v\ \{z.x > 0\}$*, formulated in the global language, states that if* u *and* v *have both positive values, then after the execution of the assignment* $z.x := u * v$*, i.e., after assigning the value of* $u * v$ *to the instance variable* x *of the object* z*, the value of* $z.x$ *is positive.*

In Chapter 5 we reformulate these Hoare triples to logical implications, using a weakest precondition calculus [Dij76, DS90] to represent the effect of assignments as in [dB99].

2.4.1 Proof outlines

For a relatively complete proof system it is necessary that the transition semantics of $Java_{seq}$ can be encoded in the assertion language. As the assertion language reasons about the local and global states, we have to *augment* the program with fresh *auxiliary variables* to represent information about the control points and stack structures within the local and global states. Invariant program properties are specified by the *annotation.* An augmented and annotated program is called a *proof outline* or an *asserted program.*

Let us dwell on the augmentation to show its motivation. Roughly speaking, the operational semantics of the programming language defines transition rules of the form[2]

$$\frac{A(T,\sigma)}{\langle T,\sigma\rangle \longrightarrow \langle T',\sigma'\rangle}\ \textsc{TransRule} \qquad ,$$

where A is an enabledness predicate over global configurations.

Soundness of a proof system means that the verification conditions assure inductivity of the annotation, i.e., its invariance under computation steps of the above form. In other words, in each reachable global configuration, the assertions attached to all current control points (and the global and class invariants, see the section on annotation below) are required to hold. Note that a single thread can stay simultaneously at several control points, one for each local configuration in its call chain. I.e., since we model method calls by synchronous communication, we need that for *every* local configuration in the call

[2] Rules of other forms are used, too, but they can be expressed in this form.

chain of a thread the associated assertion is satisfied, and not only for the local configuration on the *top* of the stack.

A proof system which ensures that the annotation is invariant under arbitrary computation steps would already be sound. But one would also wish (relative) completeness, i.e., that each invariant property is provable. Such a proof system requires that the annotation is invariant under *enabled* computation steps executed in *reachable* configurations only. That means, we must be able to *express enabledness* of computation steps in the antecedents of the verification conditions and *reachability* in the annotation. Since assertions may refer to variables only, i.e., the verification conditions argue only about the states in global configurations but not about control points and stack structures, we introduce auxiliary variables which we use to encode control information in the states. With the help of the auxiliary variables we can define a predicate $\hat{A}$ over *states* such that reachable configurations $\langle \hat{T}, \hat{\sigma} \rangle$ of the augmented program satisfy A iff the state components of $\langle \hat{T}, \hat{\sigma} \rangle$ satisfy $\hat{A}$. Note that the augmentation must not influence the original program behavior, but is only used to make observations about how a configuration is reached.

Using the predicate $\hat{A}$ we can formulate the verification conditions, which assure that the annotation is invariant under computation steps provided that the execution is enabled.

Augmentation

An augmentation extends a program by atomically executed multiple assignments $\vec{y} := \vec{e}$ to distinct auxiliary variables, which we call *observations*. Furthermore, the observations have, in general, to be "attached" to statements which they observe in an atomic manner. For object creation this is syntactically represented by the augmentation $u := \mathsf{new}^c\ \langle \vec{y} := \vec{e} \rangle^{new}$ which attaches the observation to the object creation statement. Observations $\vec{y}_1 := \vec{e}_1$ of a method call and observations $\vec{y}_4 := \vec{e}_4$ of the corresponding reception of a return value are denoted by $u := e_0.m(\vec{e})\ \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret}$. The augmentation $\langle \vec{y}_2 := \vec{e}_2 \rangle^{?call}\, stm; \mathsf{return}\ e_{ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret}$ of method bodies specifies $\vec{y}_2 := \vec{e}_2$ as the observation of the reception of the method call and $\vec{y}_3 := \vec{e}_3$ as the observation attached to the return statement. Assignments can be observed using $\vec{y} := \vec{e}\ \langle \vec{y}' := \vec{e}' \rangle^{ass}$. A stand-alone observation not attached to any statement is written as $\langle \vec{y} := \vec{e} \rangle$. It can be inserted at any point in the program.

Note that we could also use the same syntax for all kinds of observations. However, such a notation would be disadvantageous for partial augmentations, i.e., for the specification of augmentations where not all statements are observed. For example, using the notation introduced above, the augmentation $e_0.m(\vec{e})\ \langle stm \rangle$ uniquely specifies stm as a stand-alone observation following an unobserved method call; using the same augmentation syntax $\langle stm \rangle$ for all kinds of observations, we would have to write $e_0.m(\vec{e})\ \langle\rangle \langle\rangle \langle stm \rangle$ to specify the same setting. The same remark can be made also for the annotation syntax, introduced below.

The augmentation does not influence the control flow of the program but enforces a particular scheduling policy of the observations. An assignment statement and its observation are executed simultaneously. Object creation and its observation are executed in a single computation step, in this order. For method call, communication, sender, and receiver observations are executed in a single computation step, in this order (see Figure 2.1). Note that the order of the obser-

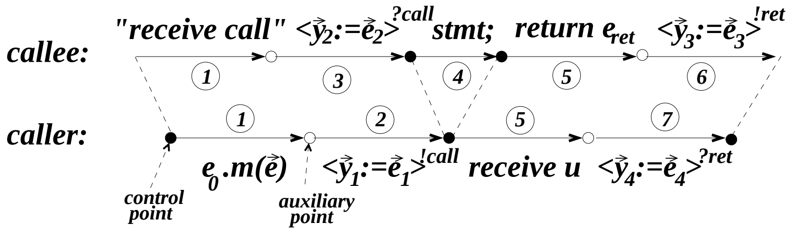


Figure 2.1: Execution order of a method call and its observations

vations plays a role for self-calls only, i.e., for method calls where the caller and the callee object are identical. Points between a statement and its observation are no *control points*, since the statement and its observation are executed in a single computation step; we call them *auxiliary points*. Note that control points are *interleaving* points, that means, while control stays at such points, other threads can execute concurrently; auxiliary points are no interleaving points.

To exclude the possibility that two observations executed in a single computation step both modify the instance state of the same object, we require that the caller observation in a self-communication may not change the values of instance variables. Without this restriction, we would have to show interference freedom under assignment-pairs, which would increase the complexity of the proof system (see Section 8.2). Formally, in each observation of a method invocation statement $e_0.m(\vec{e})$, assignments to instance variables must have the form $x := (\mathsf{if}\ e_0 = \mathsf{this}\,\mathsf{then}\ x\,\mathsf{else}\,e\,\mathsf{fi})$.

In the following we call assignment statements with their observations also multiple assignments, since they are executed simultaneously.

Similarly to program variables, in the examples we sometimes explicitly define auxiliary variables: $\langle t\ y; \rangle$ occurring in a class outside of method definitions declares y to be an auxiliary instance variable of type t. The same definition inside of a method declares y to be an auxiliary local variable of type t.

Example 2.4.2 *Extending an assignment $x := e$ to $x := e\ \langle u := x\rangle^{ass}$ stores the value of x* prior *to the execution of $x := e$ in the auxiliary variable u. Extending it to $x := e\ \langle u := x\rangle$ stores the value of x in u* after *the execution of $x := e$.*

Example 2.4.3 *We can store the number of objects created by an instance of a class c using an auxiliary integer instance variable n with initial value 0, and extending each object creation statement $u := \mathsf{new}^{c'}$ in c to $u := \mathsf{new}^{c'}\ \langle n := n+1\rangle^{new}$.*

Example 2.4.4 *We extend Example 2.4.3 by additionally observing each call* $u := e_0.m(\vec{e})$ *in* c *by* $u := e_0.m(\vec{e})\ \langle k := n\rangle^{!call}\langle k := n - k\rangle^{?ret}$. *Then the value of the auxiliary local integer variable* k *after method call and its observation, but before returning stores the number of objects created up to the call. After return, it stores the number of objects created during method evaluation.*

Example 2.4.5 *Let* l *be an auxiliary integer instance variable of a class* c. *We can count the number of local configurations executing in an instance of* c *by augmenting the body* $stm; \mathsf{return}\ e_{ret}$ *of each method in class* c *resulting in* $\langle l := l + 1\rangle^{?call}\ stm;\ \mathsf{return}\ e_{ret}\ \langle l := l - 1\rangle^{!ret}$.

The above examples show how to count objects, local configurations in an object, etc. But this information is not sufficient for a complete proof system: we have to be able to *identify* those entities. We identify a local configuration by the object in which it executes together with the value of a built-in auxiliary local variable conf storing a unique object-internal identifier. Its uniqueness is assured by the auxiliary instance variable $\mathsf{counter}$, incremented for each new local configuration in that object. The callee receives the "return address" as an auxiliary formal parameter caller of type $\mathsf{Object} \times \mathsf{Int}$, storing the identities of the caller object and the calling local configuration. The run method of the initial object is executed with the parameter caller having the value $(null, 0)$.

Syntactically, each method declaration $m(\vec{u})\{stm; \mathsf{return}\ e_{ret}\}$ gets extended by the built-in augmentation to $m(\vec{u}, \mathsf{caller})\{\langle \mathsf{conf}, \mathsf{counter} := \mathsf{counter}, \mathsf{counter} + 1\rangle^{?call} stm; \mathsf{return}\ e_{ret}\}$. Correspondingly for method calls $u := e_0.m(\vec{e})$, the actual parameter list gets extended, resulting in $u := e_0.m(\vec{e}, (\mathsf{this}, \mathsf{conf}))$. This syntactical built-in augmentation is described in more detail in Section 9.2.2. The values of the built-in auxiliary variables must not be changed by the user-defined augmentation but may be used in the augmentation and annotation. In the examples of the following sections we do not list the built-in augmentation; it is meant to be automatically included in all proof outlines.

Annotation

To specify invariant properties of the system, the augmented programs are *annotated* by attaching local assertions to each control and auxiliary point. We use the Hoare triple notation $\{p\}\ stm\ \{q\}$ and write $pre(stm)$ and $post(stm)$ to refer to the pre- and the postcondition of a statement. For assertions at auxiliary points we use the following notation: The annotation

$$\{p_0\}\ u := \mathsf{new}^c\ \{p_1\}^{new}\ \langle \vec{y} := \vec{e}\rangle^{new}\ \{p_2\}$$

of an object creation statement specifies p_0 and p_2 as pre- and postconditions, whereas p_1 at the auxiliary point should hold directly after object creation but before its observation. The annotation

$$\{p_0\}\, u := e_0.m(\vec{e}) \quad \{p_1\}^{!call}\langle \vec{y}_1 := \vec{e}_1\rangle^{!call} \quad \{p_2\}^{wait} \quad \{p_3\}^{?ret}\langle \vec{y}_4 := \vec{e}_4\rangle^{?ret} \quad \{p_4\}$$

assigns p_0 and p_4 as pre- and postconditions to the method invocation statement; p_1 is assumed to hold directly after method call, but prior to its observation; p_2 describes the control point of the caller after method call and its observation but before returning; finally, p_3 specifies the state directly after return but before its observation. The annotation of method bodies stm; $\mathsf{return}\ e_{ret}$ is defined as follows:

$$\{p_0\}^{?call}\ \langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\ \{p_1\}\quad stm;\quad \{p_2\}\ \ \mathsf{return}\ e_{ret}\ \{p_3\}^{!ret}\ \langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\ \{p_4\}$$

The callee postcondition of the method call is p_1; the callee pre- and postconditions for return are p_2 and p_4. The assertions p_0 respectively p_3 specify the states of the callee between method call, respectively, return and its observation.

Besides pre- and postconditions, for each class c, the annotation defines a local assertion I_c called *class invariant*, specifying invariant properties of instances of c in terms of its instance variables. We require that for each method of a class, the class invariant is the precondition of the method body.[3]

Finally, a global assertion GI called the *global invariant* specifies properties of communication between objects. As such, it should be invariant under object-internal computation. For that reason, we require that for all qualified references $E.x$ in GI with E of type c, all assignments to x in class c occur in the observations of communication or object creation. We require furthermore that in the annotation no free logical variables occur. In the following we will use also partially annotated statements; assertions which are not explicitly specified are by definition true.

Example 2.4.6 *The (partial) annotation* $u := \mathsf{new}^c\ \{u \neq \mathsf{this}\}$ *of an object creation statement in a class* c' *expresses that the new object's identity differs from the identity of the creator object. Invariance of this annotation can be shown by proving some verification conditions generated for the above object creation statement. However, the validity of the assertion does not depend on the rest of the program, since the only shared variable in the assertion is the self-reference, which may not be assigned to.*

The same property can be expressed using the class invariant. Since the class invariant may refer to instance variables only, we have to store the new object's identity in an auxiliary instance variable x *in order to refer to it in the class invariant. We define the annotation* $u := \mathsf{new}^c\ \langle x := u\rangle^{new}\{x = u\}$ *and the class invariant by* $x \neq \mathsf{this}$. *In this case, invariance of the given assertions depends also on the rest of the class definition: an observation* $x := \mathsf{this}$ *executed in the same object would of course violate the class invariant. This annotation is useful, if different assertions in the same class refer to* x, *and especially if the information expressed by the class invariant is needed to show properties of incoming method calls.*

Also the global invariant can be used to express the above property: Assume again $u := \mathsf{new}^c\ \langle x := u\rangle^{new}\{x = u\}$ *and let the global invariant be defined*

[3] That means, the complete annotation of method bodies is of the form $\{I_c\}\,\{p_0\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{p_1\}\ stm; \mathsf{return}\ e_{ret}\{p_3\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{p_4\}$.

by $\forall(z : c').\, z.x \neq z$. *Again, the invariance of the annotation depends on the rest of the class. But now it additionally depends on the definition of other classes, possibly creating new instances of* c', *thereby extending the domain of the quantification. Such annotations are used to express dependencies between different instance states.*

2.4.2 Verification conditions

The proof system formalizes a number of *verification conditions*, which inductively ensure that for each reachable configuration, the local assertions attached to the current control points in the thread configuration, as well as the global and the class invariants, hold. The conditions are grouped, as usual, into initial conditions, and for the inductive step into local correctness and tests for interference freedom and cooperation (see Section 1.3).

The *initial correctness* conditions cover satisfaction of the properties in the initial program configuration. The execution of a single method body in isolation is captured by standard *local correctness* conditions, using the local assertion language. Interference between concurrent method executions is covered by the *interference freedom test*, formulated also in the local language. It has especially to accommodate reentrant code. The effects of communication and object creation are treated in the *cooperation test*. As communication can take place within a single object or between different objects, the cooperation test is formulated in the global assertion language.

The verification conditions assure invariance of the annotation as follows: Initial satisfaction of the annotation is guaranteed by the initial conditions. If a computation step executes an assignment, then the local correctness conditions assure inductivity of the executing local configuration's properties; the interference freedom test assures invariance under the execution of the assignment for the properties of all other local configurations and the class invariants. For communication, invariance for the executing partners and the global invariant is assured by the cooperation test for communication. Communication itself does not affect the global state; invariance of the remaining properties under the corresponding observations is assured again by the interference freedom test. Finally for object creation, invariance for the global invariant, for the local properties of the creator, and for the created object's class invariant is assured by the conditions of the cooperation test for object creation; all other properties are invariant due to the interference freedom test.

Before specifying the verification conditions, we first introduce some notation. Let Init be a syntactical operator with interpretation *Init* (cf. page 19). Given $IVar_c$ as the set of instance variables of class c without the self-reference, and z as a logical variable of type c, let $\mathsf{InitState}(z)$ be the global assertion $z \neq \mathsf{null} \wedge \bigwedge_{x \in IVar_c} z.x = \mathsf{Init}(x)$, expressing that the object denoted by z is in its initial instance state.

Arguing about two different local configurations makes it necessary to distinguish between their local variables, since they may have the same names; in such cases we will rename the local variables in one of the local states. We use primed assertions p' to denote the given assertion p with every local variable u replaced by a fresh one u'; we use the same notation also for expressions.

Initial correctness

A proof outline of a program is *initially correct*, if the precondition of the main statement, the class invariant of the initial object, and the global invariant are satisfied initially, i.e., in the initial global configuration after the execution of the callee observation at the beginning of the main statement. Furthermore, the precondition of the observation should be satisfied prior to its execution. Since we reason about the initial global configuration, the condition for initial correctness is formulated in the global assertion language.

Definition 2.4.7 (Initial correctness) *Let the body of the* run *method of the main class c be* $\{p_2\}^{?call}\langle\vec{y}_2 := \vec{e}_2\rangle^{?call}\{p_3\}\ stm;\mathsf{return}$ *with local variables $\vec{v}$ without the formal parameters, $z \in LVar^c$, and $z' \in LVar^{\mathsf{Object}}$. A proof outline is* initially correct, *if*

$$\models_{\mathcal{G}} \quad \{\mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'\} \quad \vec{v}, \mathsf{caller} := \mathsf{Init}(\vec{v}), (\mathsf{null}, 0) \quad \{P_2(z)\}\ ,\ \textit{and} \tag{2.1}$$

$$\models_{\mathcal{G}} \quad \{\mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'\} \quad \vec{v}, \mathsf{caller} := \mathsf{Init}(\vec{v}), (\mathsf{null}, 0);\quad z.\vec{y}_2 := \vec{E}_2(z) \quad \{GI \wedge P_3(z) \wedge I_c(z)\}\ . \tag{2.2}$$

The assertion $\mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'$ states that the initial global state defines exactly one existing object z being in its initial instance state. Initialization of the local configuration is represented by the assignment $\vec{v}, \mathsf{caller} := \mathsf{Init}(\vec{v}), (\mathsf{null}, 0)$. The observation $\vec{y}_2 := \vec{e}_2$ at the beginning of the run method of the initial object z is represented by the assignment $z.\vec{y}_2 := \vec{E}_2(z)$.

Example 2.4.8 *Assume the following proof outline:*

```
{∃(z1 : Initial). z1 ≠ null ∧ ∀(z2 : Initial). z2 ≠ null → z1 = z2}  //global invariant

class Initial{
    Int x;

    {started}  //class invariant

    Void run(){
        Int v;
        ⟨Int u;⟩

        {u = 0 ∧ v = 0 ∧ x = 0}^?call  //precondition of observation
        ⟨u := 1⟩^?call                 //observation of call
        {u = 1 ∧ v = 0 ∧ x = 0}        //postcondition of observation
        ...
    }
}
```

Note that the built-in augmentation extends the observation $\{u := 1\}^{?call}$ *to* $\{u, \mathsf{started} := 1, \mathsf{true}\}^{?call}$. *The first initial condition*

$$\begin{array}{ll}\models_{\mathcal{G}} & \{z \neq \mathsf{null} \wedge z.x = 0 \wedge \forall(z' : \mathsf{Object}).\ z' = \mathsf{null} \vee z = z'\} \\ & \qquad v, u, \mathsf{caller} := 0, 0, (\mathsf{null}, 0) \\ & \qquad \{u = 0 \wedge v = 0 \wedge z.x = 0\}\end{array}$$

assures that the precondition of the observation holds after initialization but prior to its execution. The second condition

$$\begin{array}{ll}\models_{\mathcal{G}} & \{z \neq \mathsf{null} \wedge z.x = 0 \wedge \forall(z' : \mathsf{Object}).\ z' = \mathsf{null} \vee z = z'\} \\ & \quad v, u, \mathsf{caller} := 0, 0, (\mathsf{null}, 0); \quad u, z.\mathsf{started} := 1, \mathsf{true} \\ & \quad \{GI \wedge (u = 1 \wedge v = 0 \wedge x = 0) \wedge (z.\mathsf{started})\}\end{array}$$

assures that the global invariant, the postcondition of the observation, and the class invariant hold after the observation. Satisfaction of the global invariant can be shown by instantiation with z. *We use this example also in Section 9.2.4 to illustrate the usage of the* Verger *tool.*

Local correctness

A proof outline is *locally correct*, if the properties of method instances as specified by the annotation are invariant under their own execution, i.e., if the usual verification conditions [Apt81b] for standard sequential constructs hold. For example, the precondition of an assignment must imply its postcondition after its execution. Besides conditions for assignments, local correctness defines additional conditions for control structures like loops and conditional statements. The following condition should hold for all multiple assignments being an assignment statement with its observation, an unobserved assignment, or a stand-alone observation:

Definition 2.4.9 (Local correctness: Assignment) *A proof outline is* locally correct *with respect to assignments, if for all multiple assignments* $\{p_1\}\,\vec{y} := \vec{e}\,\{p_2\}$ *in class* c, *which are not the observation of object creation or communication,*

$$\models_{\mathcal{L}} \quad \{p_1 \wedge I_c\} \quad \vec{y} := \vec{e} \quad \{p_2\}\ . \tag{2.3}$$

Prior to the execution of the assignment $\vec{y} := \vec{e}$, the assertion attached to the current control point of the executing local configuration, i.e., the precondition of the assignment, is required to hold. Execution causes the control to move to the point after the assignment. Thus the assertion at the new control point, i.e., the postcondition of the assignment, should hold after execution.

We use the class invariant as antecedent whose invariance is assured by the interference freedom test. Note that including the class invariant as antecedent in the local correctness conditions is not necessary for a minimal proof system, since the class invariant itself can be stated in the local assertions, too. However, it reduces the annotation. The same holds for the interference freedom and cooperation tests.

The conditions for loops and conditional statements are standard. Note that we have no local verification conditions for observations of communication and object creation. The postconditions of such statements express *assumptions* about the communicated values. These assumptions will be verified in the *cooperation test*.

Example 2.4.10 *Assume the following augmented and annotated method which computes the faculty* $u!$ *for its parameter* u*:*

```
Int fac(Int u){
     Int result;
     {u > 0}
     result:=1; {result = 1 ∧ u > 0}
     v:=u; {u! = result ∗ v! ∧ u > 0 ∧ v > 0}
     while (v>1) do {u! = result ∗ v! ∧ u > 0 ∧ v > 1}
          result:=result*v; {u! = result ∗ (v − 1)! ∧ u > 0 ∧ v > 1}
          v:=v-1; {u! = result ∗ v! ∧ u > 0 ∧ v > 0}
     od;  {u! = result}
     return result
}
```

The above proof outline satisfies the conditions of local correctness. There are 7 local correctness conditions (there are no initial correctness, interference freedom, and cooperation test conditions for this example). For example, for the assignment $\mathsf{result} := \mathsf{result} * v$ *local correctness defines the verification condition*

$$\models_{\mathcal{L}} \quad \{u! = \mathsf{result} * v! \wedge u > 0 \wedge v > 1\} \quad \mathsf{result} := \mathsf{result} * v \quad \{u! = \mathsf{result} * (v-1)! \wedge u > 0 \wedge v > 1\}\ ,$$

whose satisfaction is easy to see.

The interference freedom test

Interference between concurrent method executions is covered by the proof obligations of the *interference freedom test*. Since we are dealing with a sequential language, we only need to show invariance of assertions attached to control points waiting for return in a call chain under execution of the local configuration on the top of the stack. Interference freedom covers also invariance of the class invariants.

Since $Java_{seq}$ does not support qualified references to instance variables, execution in an object cannot influence the evaluation of local assertions in other objects. That means, we only have to deal with invariance under execution within the *same* object. Therefore, the corresponding verification conditions are formulated in the local assertion language. Affecting only local variables, communication and object creation do not change the instance states of the executing objects[4]. Thus we only have to cover invariance of assertions at control points under *assignments*, including observations of communication and object

[4] It is due to the restriction that method call and object creation statements may not contain instance variables.

creation. To distinguish local variables of the different local configurations, we rename those of the assertion. Note that assertions at auxiliary points do not have to be shown invariant, since auxiliary points are no interleaving points.

Let q be an assertion at a control point and $\vec{y} := \vec{e}$ a multiple assignment in the same class c. In which cases does q have to be invariant under the execution of the assignment? Since the language is sequential, i.e., q and $\vec{y} := \vec{e}$ belong to the *same* thread, the only assertions endangered are those at control points waiting for return earlier in the current execution stack. Invariance of a local configuration under its own execution, however, does not need to be considered and is excluded by requiring $\mathsf{conf} \neq \mathsf{conf}'$. For an assertion at a control point waiting for returning from a self-call, interference with the *matching* return statement needs neither be considered: The communicating partners execute simultaneously changing also the control point of the caller. The assertion $\mathsf{caller} = (\mathsf{this}, \mathsf{conf}')$ describes this setting: It holds if the local configuration described by q' and the identity conf' is the caller of the local configuration with local variable caller which executes $\vec{y} := \vec{e}$ in the same object. Let $\mathsf{caller\_obj}$ be the first and $\mathsf{caller\_conf}$ the second component of caller. We define $\mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e})$ by

- $\mathsf{conf}' \neq \mathsf{conf}$, for assertions $\{q\}^{wait}$ attached to control points waiting for return, if $\vec{y} := \vec{e}$ is not the observation of return;
- $\mathsf{conf}' \neq \mathsf{conf} \wedge (\mathsf{this} \neq \mathsf{caller\_obj} \vee \mathsf{conf}' \neq \mathsf{caller\_conf})$, for assertions $\{q\}^{wait}$, if $\vec{y} := \vec{e}$ observes return;
- false, otherwise.

For the example configuration intuitively shown in Figure 2.2, the assertion p_3, attached to a control point waiting for return, has to be invariant under the execution of the assignment by its callee, while p_4 does not have to be invariant under its own execution. However, if the assignment would be the callee observation of a return statement, then p_3, describing the communication partner, would not have to be invariant under the assignment. The assertions p_1 and p_2 are automatically invariant, since they describe an object different from the one in which the execution takes place. Note that satisfaction of p_5 after execution is assured by the local correctness conditions.
The interference freedom test can now be formulated as follows:

Definition 2.4.11 (Interference freedom) *A proof outline is* interference free, *if for all classes c and multiple assignments $\vec{y} := \vec{e}$ with precondition p in c,*

$$\models_{\mathcal{L}} \quad \{p \wedge I_c\} \quad \vec{y} := \vec{e} \quad \{I_c\}\ . \tag{2.4}$$

Furthermore, for all assertions q at control points in c,

$$\models_{\mathcal{L}} \quad \{p \wedge q' \wedge I_c \wedge \mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e})\} \quad \vec{y} := \vec{e} \quad \{q'\}\ . \tag{2.5}$$

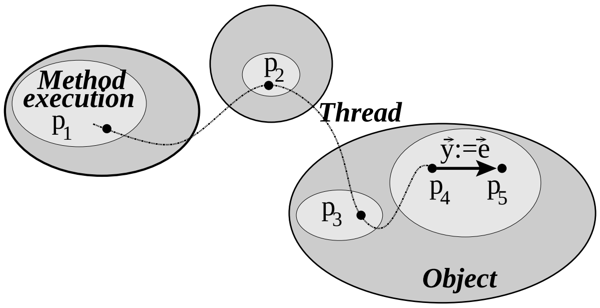


Figure 2.2: Interference for a single thread

Note that if we would allow qualified references in program expressions, we would have to show interference freedom for all assertions under all assignments in programs, not only for those occurring in the same class. For a program with n classes where each class contains k assignments and l assertions at control points, the number of interference freedom conditions is in $\mathcal{O}(c \cdot k \cdot l)$, instead of $\mathcal{O}((c \cdot k) \cdot (c \cdot l))$ with qualified references.

Example 2.4.12 *Let* $\{p_1\}\,\mathsf{this}.m(\vec{e})\ \{p_2\}^{!call}\langle stm_1\rangle^{!call}\{p_3\}^{wait}\{p_4\}^{?ret}\langle stm_2\rangle^{?ret}\{p_5\}$ *be an annotated method call statement in a method m' of a class c with an integer auxiliary instance variable x, such that each assertion implies* $\mathsf{conf} = x$. *I.e., the identity of the executing local configuration is stored in the instance variable x. The annotation expresses that no pairs of control points in m' of c can be simultaneously reached.*

The assertions p_2 and p_4 need not be shown invariant, since they are attached to auxiliary points. Interference freedom neither requires invariance of the assertions p_1 and p_5, since they are not at control points waiting for return, and thus the antecedents of the corresponding conditions evaluate to false. Invariance of p_3 under the execution of the observation stm_1 with precondition p_2 requires validity of $\models_{\mathcal{L}} \{p_2 \wedge p_3' \wedge \mathsf{waits\_for\_ret}(p_3, stm_1)\}\ stm_1\ \{p_3'\}$. *The assertion* $p_2 \wedge p_3' \wedge \mathsf{waits\_for\_ret}(p_3, stm_1)$ *implies* $(\mathsf{conf} = x) \wedge (\mathsf{conf}' = x) \wedge (\mathsf{conf}' \neq \mathsf{conf})$, *which evaluates to false. Invariance of p_3 under stm_2 follows analogously.*

Example 2.4.13 *Assume a partially*[5] *annotated method invocation statement of the form* $\{p_1\}\,\mathsf{this}.m(\vec{e})\ \{\mathsf{conf} = x \wedge p_2\}^{wait}\{p_3\}$ *in a class c with an integer auxiliary instance variable x, and assume that method m of c has the annotated return statement* $\{q_1\}\ \mathsf{return}\ \{\mathsf{caller} = (\mathsf{this}, x)\}^{!ret}\langle stm\rangle^{!ret}\{q_2\}$. *The annotation expresses that the local configurations containing the above statements are in caller-callee relationship. Thus upon return, the control point of the caller moves from the point at* $\mathsf{conf} = x \wedge p_2$ *to that at p_3, i.e.,* $\mathsf{conf} = x \wedge p_2$ *does not have to be invariant under the observation of the return statement.*

Again, the assertion $\mathsf{caller} = (\mathsf{this}, x)$ *at an auxiliary point does not have to be shown invariant. For the assertions p_1, p_3, q_1, and q_2, which are not at a control*

[5] As already mentioned, missing assertions are by definition true.

point waiting for return, the antecedent is false. Invariance of $\mathsf{conf} = x \wedge p_2$ *under the observation stm with precondition* $\mathsf{caller} = (\mathsf{this}, x)$ *is covered by the interference freedom condition*

$$\models_{\mathcal{L}} \quad \{ \quad \mathsf{caller} = (\mathsf{this}, x) \wedge (\mathsf{conf}' = x \wedge p_2')\wedge \\ \mathsf{waits\_for\_ret}((\mathsf{conf} = x \wedge p_2), stm) \quad \} \quad stm \quad \{\mathsf{conf}' = x \wedge p_2'\} \; .$$

The $\mathsf{waits\_for\_ret}$ *assertion implies* $\mathsf{caller} \neq (\mathsf{this}, \mathsf{conf}')$*, which contradicts the assumptions* $\mathsf{caller} = (\mathsf{this}, x)$ *and* $\mathsf{conf}' = x$*; thus the antecedent of the condition is false.*

Satisfaction of $\mathsf{conf} = x \wedge p_2$ *after the call, satisfaction of* $\mathsf{caller} = (\mathsf{this}, x)$ *directly after return, and satisfaction of* p_3 *and* q_2 *after the observation stm is assured by the cooperation test.*

The cooperation test

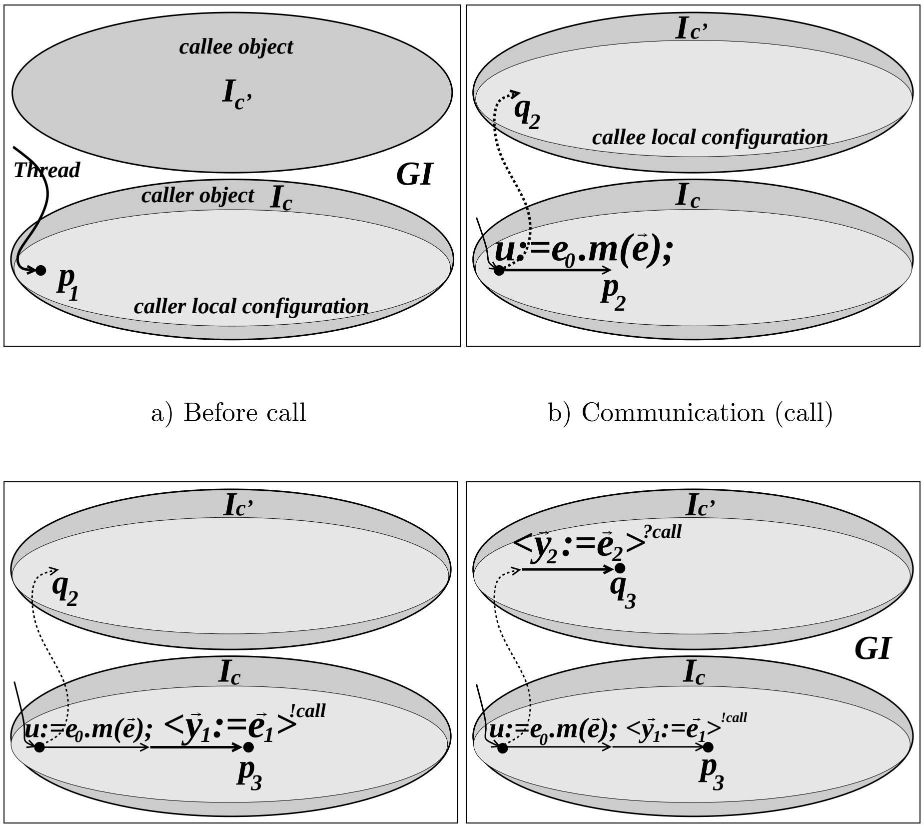
Whereas the interference freedom test assures invariance of assertions under steps in which they are not involved, the *cooperation test* deals with inductivity for communicating partners, assuring that the global invariant, and the preconditions and the class invariants of the involved statements imply their postconditions after the joint step. Additionally, the preconditions of the corresponding observations must hold immediately after communication.

The global invariant refers to auxiliary instance variables which can be changed by observations of communication, only. Consequently, the global invariant is automatically invariant under the execution of non-communicating statements. For communication and object creation, however, the invariance must be shown as part of the cooperation test.

We start with the cooperation test for method invocation. The semantics of method call and returning from a method is intuitively shown in Figures 2.3 and 2.4. After communication, i.e., after creating and initializing the callee local configuration and passing on the actual parameters (2.3 b), first the caller (2.3 c), and then the callee (2.3 d) execute their corresponding observations, all in a single computation step. Correspondingly for return, after communicating the result value (2.4 f), first the callee (2.4 g) and then the caller observation (2.4 h) gets executed.

To avoid name clashes between local variables of the partners, we rename those of the callee. Since different objects may be involved, the cooperation test is formulated in the global assertion language. Local properties are expressed in the global language using the lifting substitution. As already mentioned, we use the shortcuts $P(z)$ and $Q'(z')$ for $p[z/\mathsf{this}]$ and for $q'[z'/\mathsf{this}]$, respectively, and similarly for expressions.

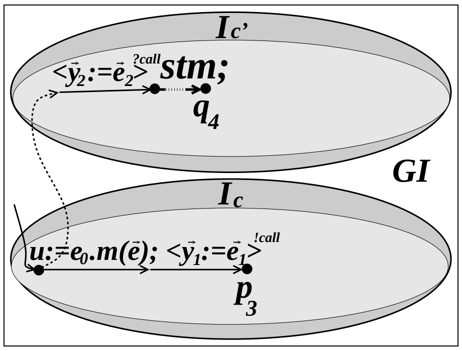
Let z and z' be logical variables whose values represent the caller, respectively, the callee object in a method call. We assume the global invariant, the class invariants of the communicating partners, and the preconditions of the communicating statements to hold prior to communication. For method invocation, the precondition of the callee is its class invariant. That the two statements indeed represent communicating partners is captured by the assertion



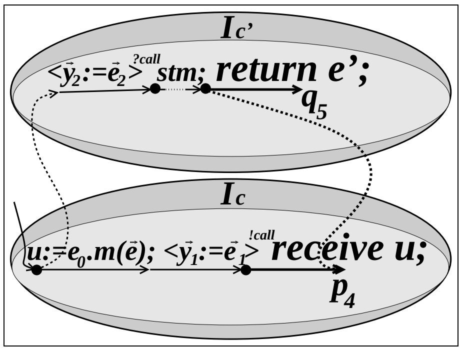
a) Before call b) Communication (call)

c) Caller observation d) Callee observation

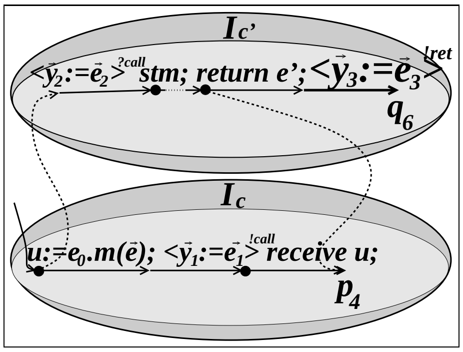
Figure 2.3: Execution of a method call $\{p_1\}\, u := e_0.m(\vec{e})\ \{p_2\}^{!call}\langle\vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_3\}^{wait}$ with callee method body $\{q_2\}^{?call}\langle\vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\, stm; \mathsf{return}\ e'$. Control points are marked by a dot.



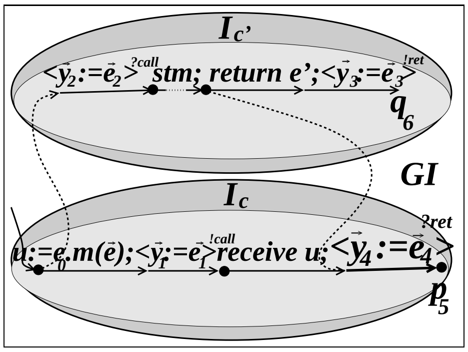
e) Method evaluation



f) Communication (return)



g) Callee observation



h) Caller observation

Figure 2.4: Execution of return for a method call
$\{p_1\}\, u := e_0.m(\vec{e})\ \{p_2\}^{!call}\, \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call}\, \{p_3\}^{wait}\, \{p_4\}^{?ret}\, \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret}\, \{p_5\}$
with callee method body
$\{q_2\}^{?call}\, \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call}\, \{q_3\}\ stm; \{q_4\}$ return e' $\{q_5\}^{!ret}\, \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret}\, \{q_6\}$.
Control points are marked by a dot.

comm, which depends on the type of communication: For method invocation $e_0.m(\vec{e})$, the assertion $E_0(z) = z'$ states that the value of z' indeed identifies the callee object. Remember that method invocation hands over the return address as an auxiliary parameter, and that the values of formal parameters remain unchanged. Furthermore, actual parameters may not contain instance variables, i.e., their interpretation does not change during method execution. Therefore, the formal and actual parameters can be used at returning from a method to identify partners being in caller-callee relationship, using the built-in auxiliary variables. Thus for the return case, comm additionally states $\vec{u}' = \vec{E}(z)$, where $\vec{u}$ and $\vec{e}$ are the formal and the actual parameters. Returning from the run method terminates the executing thread; this does not have communication effects.

As in the previous conditions, state changes are expressed by assignments. For the example of method invocation, communication is expressed by the assignment $\vec{u}' := \vec{E}(z)$, where initialization of the remaining local variables $\vec{v}$ is covered by $\vec{v}' := \mathsf{Init}(\vec{v})$. The assignments $z.\vec{y}_1 := \vec{E}_1(z)$ and $z'.\vec{y}'_2 := \vec{E}'_2(z')$ stand for the caller and callee observations $\vec{y}_1 := \vec{e}_1$ and $\vec{y}_2 := \vec{e}_2$, executed in the objects z and z', respectively. Note that we rename all local variables of the callee to avoid name clashes.

Definition 2.4.14 (Cooperation test: Communication) *A proof outline satisfies the* cooperation test for communication*, if*

$$\models_{\mathcal{G}} \quad \{GI \wedge P_1(z) \wedge I_c(z) \wedge Q'_1(z') \wedge I_{c'}(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\} \; f_{comm} \; \{P_2(z) \wedge Q'_2(z')\} \; \text{and} \tag{2.6}$$

$$\models_{\mathcal{G}} \quad \{GI \wedge P_1(z) \wedge I_c(z) \wedge Q'_1(z') \wedge I_{c'}(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\} \; f_{comm}; \; f_{obs1}; \; f_{obs2} \; \{GI \wedge P_3(z) \wedge Q'_3(z')\} \tag{2.7}$$

hold for distinct fresh logical variables $z \in LVar^c$ and $z' \in LVar^{c'}$, in the following cases:

1. CALL*: For all statements $\{p_1\}\, u_{ret} := e_0.m(\vec{e})\; \{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1\rangle^{!call} \{p_3\}^{wait}$ (or such without receiving a value) in class c with e_0 of type c', where method m of c' has body $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2\rangle^{?call} \{q_3\}\, stm; \mathsf{return}\ e_{ret}$, formal parameters $\vec{u}$, and local variables $\vec{v}$ except the formal parameters. The callee class invariant is $q_1 = I_{c'}$. The assertion* comm *is given by $E_0(z) = z'$. Furthermore, f_{comm} is $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$, f_{obs1} is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{obs2} is $z'.\vec{y}'_2 := \vec{E}'_2(z')$.*

2. RETURN*: For all $u_{ret} := e_0.m(\vec{e})\ \langle stm\rangle^{!call} \{p_1\}^{wait} \{p_2\}^{?ret} \langle \vec{y}_4 := \vec{e}_4\rangle^{?ret} \{p_3\}$ (or such without receiving a value) occurring in c with e_0 of type c', such that method m of c' has the return statement $\{q_1\}$ $\mathsf{return}\ e_{ret}$ $\{q_2\}^{!ret} \langle \vec{y}_3 := \vec{e}_3\rangle^{!ret} \{q_3\}$, and formal parameter list $\vec{u}$, the above equations must hold with* comm *given by $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$, and where f_{comm} is $u_{ret} := E'_{ret}(z')$, f_{obs1} is $z'.\vec{y}'_3 := \vec{E}'_3(z')$, and f_{obs2} is $z.\vec{y}_4 := \vec{E}_4(z)$.*

3. $\textsc{Return}_{run}$: *For* $\{q_1\}$ return $\{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$ *occurring in the* run *method of the main class,* $p_1 = p_2 = p_3 =$ true, comm $=$ true, *and furthermore* f_{comm} *and* f_{obs2} *are the empty statement, and* f_{obs1} *is* $z'.\vec{y}'_3 := \vec{E}'_3(z')$.

Example 2.4.15 *This example illustrates how one can prove properties of parameter passing. Let* $\{p\}\, e_0.m(v, \vec{e})$, *with* p *given by* $v > 0$, *be a (partially) annotated statement in a class* c *with* e_0 *of type* c', *and let method* $m(u, \vec{w})$ *of* c' *have a body of the form* $\{q\}\, stm;$return *where* q *is* $u > 0$. *Inductivity of the proof outline requires that if* p *is valid prior to the call (besides validity of the global and class invariants), then* q *is satisfied after the invocation. Omitting irrelevant details, Condition 2.7 of the cooperation test requires proving* $\models_{\mathcal{G}} \{P(z)\}\ u' := v\ \{Q'(z')\}$, *which expands to* $\models_{\mathcal{G}} \{v > 0\}\ u' := v\ \{u' > 0\}$.

Example 2.4.16 *The following example demonstrates how one can express dependencies between instance states in the global invariant and use this information in the cooperation test.*

Let $\{p\}\, e_0.m(\vec{e})$, *with* p *given by* $x > 0 \wedge e_0 = o$, *be an annotated statement in a class* c *with* e_0 *of type* c', x *an integer instance variable, and* o *an instance variable of type* c', *and let method* $m(\vec{u})$ *of* c' *have the annotated body* $\{q\}\, stm;$return *where* q *is* $y > 0$ *and* y *an integer instance variable. Let furthermore* $z \in LVar^c$ *and let the global invariant be given by* $\forall z.\,(z \neq \mathsf{null} \wedge z.o \neq \mathsf{null} \wedge z.x > 0) \rightarrow z.o.y > 0$. *Inductivity requires that if* p *and the global invariant are valid prior to the call, then* q *is satisfied after the invocation (again, we omit irrelevant details). The cooperation test Condition 2.7, i.e.,* $\models_{\mathcal{G}} \{GI \wedge P(z) \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\}\ \vec{u}' := \vec{E}(z)\ \{Q'(z')\}$ *expands to*

$$\begin{array}{ll}\models_{\mathcal{G}} & \{(\forall z.\,(z \neq \mathsf{null} \wedge z.o \neq \mathsf{null} \wedge z.x > 0) \rightarrow z.o.y > 0) \wedge \\ & \quad (z.x > 0 \wedge E_0(z) = z.o) \wedge E_0(z) = z' \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\ \} \\ & \qquad\qquad \vec{u}' := \vec{E}(z) \\ & \qquad\qquad \{z'.y > 0\}\ .\end{array}$$

Instantiating the quantification by z, *the antecedent implies* $z.o.y > 0 \wedge z' = z.o$, *i.e.,* $z'.y > 0$. *Invariance of the global invariant is straightforward.*

Example 2.4.17 *This example illustrates how the cooperation test handles observations of communication. Let* $\{\neg b\}\,\mathsf{this}.m(\vec{e})\{b\}^{wait}$ *be an annotated statement in a class* c *with boolean auxiliary instance variable* b *and let* $m(\vec{u})$ *of* c *have a body of the form* $\{\neg b\}^{?call}\langle b := \mathsf{true}\rangle^{?call}\{b\}\, stm;$return. *Condition 2.6 of the cooperation test assures inductivity for the precondition of the observation. We have to show* $\models_{\mathcal{G}} \{\neg z.b \wedge \mathsf{comm}\}\vec{u}' := \vec{E}(z)\{\neg z'.b\}$ *(again, we omit irrelevant details), i.e., since it is a self-call,* $\models_{\mathcal{G}} \{\neg z.b \wedge z = z'\}\vec{u}' := \vec{E}(z)\{\neg z'.b\}$, *which is trivially satisfied. Condition 2.7 of the cooperation test for the postconditions requires* $\models_{\mathcal{G}} \{\mathsf{comm}\}\vec{u}' := \vec{E}(z); z'.b := \mathsf{true}\{z.b \wedge z'.b\}$ *which expands to* $\models_{\mathcal{G}} \{z = z'\}\vec{u}' := \vec{E}(z); z'.b := \mathsf{true}\{z.b \wedge z'.b\}$, *whose validity is easy to see.*

Besides method calls and returns, the cooperation test needs to handle object creation, taking care of the preservation of the global invariant, the postcondition of the new-statement and its observation, and the new object's class invariant. We can assume that the precondition of the object creation statement, the class invariant of the creator, and the global invariant hold in the configuration prior to instantiation. The extension of the global state with a freshly created object is formulated in a *strongest postcondition* style, i.e., it is required to hold immediately *after* the instantiation. We use existential quantification to refer to the old value: z' of type $LVar^{\mathsf{list\ Object}}$ represents the existing objects prior to the extension. Moreover, that the created object's identity stored in u is fresh and that the new instance is properly initialized is expressed by the global assertion $\mathsf{Fresh}(z', u)$ defined as $\mathsf{InitState}(u) \wedge u \notin z' \wedge \forall(v : \mathsf{Object}).\ v \in z' \vee v = u$ (see page 32 for the definition of InitState). To express that an assertion refers to the set of existing objects *prior* to the extension of the global state, we need to *restrict* any quantification in the assertion to range over objects from z', only. So let P be a global assertion and $z' \in LVar^{\mathsf{list\ Object}}$ a logical variable not occurring in P. Then $P \downarrow z'$ is the global assertion P with all quantifications $\exists z.\ P'$ replaced by $\exists z.\ \mathsf{obj}(z) \subseteq z' \wedge P'$, where $obj(v)$ denotes the set of objects occurring in the value v. The following lemma formulates the basic property of the projection operator:

Lemma 2.4.18 *Assume a global state σ, an extension $\sigma' = \sigma[\alpha \mapsto \sigma_{inst}^{c,init}]$ for some $\alpha \in Val^c$, $\alpha \notin Val(\sigma)$, and a logical environment ω referring only to values existing in σ. Let v be the sequence consisting of all elements of $\bigcup_c Val_{null}^c(\sigma)$. Then for all global assertions P and logical variables $z' \in LVar^{\mathsf{list\ Object}}$ not occurring in P,*

$$\omega, \sigma \models_{\mathcal{G}} P \quad \textit{iff} \quad \omega[z' \mapsto v], \sigma' \models_{\mathcal{G}} P \downarrow z'.$$

Its proof can be found in Appendix A.1. Thus a predicate $(\exists u.\ P) \downarrow z'$, evaluated immediately after the instantiation $u := \mathsf{new}^c$, expresses that P holds prior to the creation of the new object. This leads to the following definition of the cooperation test for object creation.

Definition 2.4.19 (Cooperation test: Instantiation) *A proof outline satisfies the* cooperation test for object creation*, if for all classes c' and statements $\{p_1\}\, u := \mathsf{new}^c\ \{p_2\}^{new} \langle \vec{y} := \vec{e} \rangle^{new} \{p_3\}$ in c':*

$$\models_{\mathcal{G}} \quad z {\neq} \mathsf{null} \wedge z {\neq} u \wedge \exists z'.\ \big(\mathsf{Fresh}(z', u) \wedge (GI \wedge (\exists u.\ P_1(z)) \wedge I_{c'}(z)) \downarrow z'\big) \\ \rightarrow P_2(z) \wedge I_c(u) \ \textit{and} \tag{2.8}$$

$$\models_{\mathcal{G}} \quad \{z {\neq} \mathsf{null} \wedge z {\neq} u \wedge \exists z'.\ \big(\mathsf{Fresh}(z', u) \wedge (GI \wedge (\exists u.\ P_1(z)) \wedge I_{c'}(z)) \downarrow z'\big)\} \\ z.\vec{y} := \vec{E}(z) \\ \{GI \wedge P_3(z)\} \tag{2.9}$$

hold with $z \in LVar^{c'}$ and $z' \in LVar^{\mathsf{list\ Object}}$ fresh.

Example 2.4.20 *Assume a statement* $u := \mathsf{new}^c\{u \neq \mathsf{this}\}$ *in a program, where the class invariant of* c *is* $x \geq 0$ *for an integer instance variable* x*. Condition 2.8 of the cooperation test for object creation assures that the class invariant of the new object holds after its creation. We have to show validity of* $\models_{\mathcal{G}} (\exists z'.\, \mathsf{Fresh}(z', u)) \to u.x \geq 0$*, i.e.,* $\models_{\mathcal{G}} u.x = 0 \to u.x \geq 0$*, which is trivial. Remember that integer variables have the initial value* 0*. For the postcondition, Condition 2.9 requires* $\models_{\mathcal{G}} \{z \neq u\}\ \epsilon\ \{u \neq z\}$ *with* ϵ *the empty statement (no observations are executed), which is true.*

Example 2.4.21 *Assume now a statement* $u := \mathsf{new}^c\{u \neq x\}$ *in a class* c' *with instance variable* x *of type* c*, where the class invariants are* true*, and the global invariant is* $\forall (z_1 : c').\, z_1 \neq \mathsf{null} \to \exists (z_2 : c) : z_1.x = z_2$*. Condition 2.9 requires*

$$\begin{array}{ll} \models_{\mathcal{G}} & \{z \neq \mathsf{null} \wedge z \neq u \wedge \\ & \exists z'.\, \big(u \neq \mathsf{null} \wedge u \notin z' \wedge (\forall v.\, v \in z' \vee v = u) \wedge GI \downarrow z'\big)\} \\ & \epsilon \quad \{u \neq z.x\}\,, \end{array}$$

where ϵ *is again the empty statement. Now, the antecedent implies that there is a sequence* z' *of objects such that* $u \notin z'$*. Furthermore, from* $z \neq u$ *and from* $\forall v.\, v \in z' \vee v = u$ *we conclude that* $z \in z'$*. The assertion* $GI \downarrow z'$ *is given by*

$$(\forall (z_1 : c').\, z_1 \neq \mathsf{null} \to \exists (z_2 : c) : z_1.x = z_2) \downarrow z'\,,$$

i.e.,

$$\forall (z_1 : c').\, z_1 \in z' \to z_1 \neq \mathsf{null} \to \exists (z_2 : c) : z_2 \in z' \wedge z_1.x = z_2\,.$$

Instantiating the above assertion with z *we get that* $z_2 \in z' \wedge z.x = z_2$ *for some* z_2*, i.e,* $z.x \in z'$*. Since* $u \notin z'$*, it implies that* $z.x \neq u$*, as required.*

In the example above we used a tautology $\forall (z_1 : c').\, z_1 \neq \mathsf{null} \to \exists (z_2 : c) : z_1.x = z_2$ as global invariant in order to prove the required property $u \neq x$ of the creator. This was necessary since the assertion $\mathsf{Fresh}(z', u)$ expresses only that z' is the sequence of all existing objects without u and that u is in its initial state, but not that u is fresh in the sense that no variables refer to it in the states prior to its creation. However, since the verification condition assumes $GI \downarrow z'$, the above tautology as global invariant restricted to z' expresses this missing information for the instance variable x of the creator!

With an alternative definition of the assertion $\mathsf{Fresh}(z', u)$ which would additionally state

$$\left(\bigwedge_{v \in TVar \setminus \{u\}} v \neq u \right) \wedge \left(\forall (z : \mathsf{Object}).\ z \neq \mathsf{null} \to \bigwedge_{x \in IVar(z)} z.x \neq u \right)$$

for the set $TVar$ of local variables —to be precise, we need only those of the creator local configuration—, and the set $IVar(z)$ of instance variables of objects z, we could prove properties like $x \neq u$ above without additional information. However, this information is not needed for a minimal proof system, as demonstrated by the above example.

Examples

Example 2.4.22 *The following proof outline computes the integer division* `i` *of two natural numbers* `n` *and* `d`*, as stated by the annotation:*

```
class IntegerDivision{
    Void run(){
        Int n,d,i;
        ...
        {n ≥ 0 ∧ d > 0}
        i := m(n,d); {n ≥ 0 ∧ d > 0}^wait  {i * d ≤ n ∧ n < (i + 1) * d}
        ...
    }

    Int m(Int n, Int d){
        Int u,i;

        {n ≥ 0}
        u := n; {u = n ∧ u ≥ 0}
        i := 0; {n = i * d + u ∧ i ≥ 0 ∧ u ≥ 0}
        while (u ≥ d) do
            {n = i * d + u ∧ i ≥ 0 ∧ u ≥ 0 ∧ u ≥ d}
            u := u-d; {n = i * d + u + d ∧ i ≥ 0 ∧ u ≥ 0}
            i := i+1; {n = i * d + u ∧ i ≥ 0 ∧ u ≥ 0}
        od;
        {n = i * d + u ∧ i ≥ 0 ∧ u ≥ 0 ∧ u < d}
        return i
    }
}
```

We have 7 local conditions, all for statements in the method `m`*, and two global conditions for the invocation of and for returning from the method* `m`*. Since the only assertion at a control point waiting for return contains local variables only, its invariance under execution is easy to see; all interference freedom conditions are trivial. All conditions have been automatically proven in the theorem prover PVS.*

Example 2.4.23 *The following program consists of a single main class with instance variables* `x` *and* `y`*, an auxiliary instance variable* `at`*, and class invariant* `x≥0`*. The class declares two methods* `run` *and* `m`*. The method* `m` *simply decrements the value of* `x` *by the value of* `y`*. The* `run` *method invokes* `m` *in case* `x≥y`*. To express that after the invocation the new value of* `x` *is the old value minus* `y`*, we store the old value in the auxiliary local variable* `v`*.*

```
class Annotation{
    Int x,y;
    ⟨Int at;⟩ //auxiliary instance variable
    {x ≥ 0}   //class invariant

    Void run(){
        ⟨Int v;⟩ //auxiliary local variable
        ...
        {at = 0}
        if (x≥y) then
            {x ≥ y ∧ at = 0}
            m(); ⟨v := x⟩^!call  {(at = 1 ∧ x = v) ∨ (at = 2 ∧ x = v − y)}^wait
            {x = v − y}
        fi
        ...
    }
```

```
    Void m(){
        {at = 0}^?call  ⟨at := 1⟩^?call  {at = 1 ∧ x ≥ y}
        x:=x-y; ⟨at := 2⟩^ass  {at = 2}
        return
    }
}
```

We have two local conditions, one for entering the body of the conditional `if`*-statement, and one for the multiple assignment* `x:=x-y;` $\langle at := 2; \rangle^{ass}$ *in* `m`*. We have* 6 *interference freedom conditions: One interference freedom condition is generated for the invariance of the class invariant under the assignment* `x:=x-y` *with its observation in* `m`*, and two for the invariance of the assertion at the control point waiting for return in* `run` *under the above assignment and under the observation* $\langle at := 1; \rangle^{?call}$ *in* `m`*. The remaining interference freedom conditions are trivial. Two cooperation test conditions take care for the properties of the method call and the corresponding return. All conditions have been automatically proven in the theorem prover PVS.*

Example 2.4.24 *Assume the following class containing an annotated method which computes the faculty* $u!$ *of its parameter* u*, similarly to Example 2.4.10 but now using recursive method calls:*

```
class FacRec{
    Int fac(Int u){
        Int v;
        {u > 0}
        if (u ≤ 1) then {u = 1}
            v := 1; {u > 0 ∧ v = u!}
        else {u > 1}
            v := fac(u-1); {u > 1 ∧ v = (u − 1)!}
            v := u*v;   {u > 1 ∧ v = u!}
        fi {u > 0 ∧ v = u!}
        return v
    }
}
```

The class and global invariants are by definition true. For the above proof outline 8 *verification conditions are generated (*6 *local correctness conditions and* 2 *cooperation test conditions for calling and returning from the method* fac*). All conditions are verified automatically in PVS using the* grind *strategy. Note that since the method does not refer to instance variables, no interference freedom conditions are generated. Note furthermore that for this example no augmentation is needed.*

The local conditions are straightforward. For the call $\mathsf{v} := \mathsf{fac}(\mathsf{u}-1)$ *the cooperation test Condition 2.7 requires*

$$\models_{\mathcal{G}} \quad \{u > 1 \wedge z = z' \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\} \quad u' := u - 1 \quad \{u' > 0\}\,.$$

Note that, since no instance variables are involved, the above global condition is equivalent to the local *condition*

$$\models_{\mathcal{L}} \quad \{u > 1 \wedge z = z' \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\} \quad u' := u - 1 \quad \{u' > 0\}\,.$$

For the corresponding return case the cooperation test requires according to Condition 2.7

$$\models_{\mathcal{G}} \quad \{u' > 0 \wedge v' = u'! \wedge z = z' \wedge u' = u - 1 \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\} \quad v := v' \\ \{u > 1 \wedge v = (u-1)!\}\,,$$

whose validity is easy to see. Also this condition does not refer to instance variables, and thus it is also equivalent to a local condition.

Example 2.4.25 *Assume the following recursive method which returns the value of the integer instance variable x at the time of its invocation:*

```
Int m() {
    Int v;
    if (x>0) then
        x:=x-1;
        v:=m();
        x:=v+1;
    fi;
    return x
}
```

We would like to prove the property that the return value of the method is the value of x at the time of its invocation. To express this requirement, we can store the value of x at invocation in an auxiliary integer local variable u by inserting the callee observation $\langle u := x\rangle^{?call}$ of the call, and define $u = x$ as the precondition of the return statement. To be able to define a proof outline which satisfies the verification conditions and implies the above annotation we need to encode properties of the recursive invocations in sequences, which is possible but quite complex.

However, allowing also user-defined auxiliary parameters *would lead to a much natural and simpler solution, listed below. Such an extension of the augmentation is straightforward and does not require any modification of the verification conditions.*

```
Int m(int u) {
    Int v;
    {u = x}
    if (x>0) then {u = x}
        x:=x-1; {u = x + 1}
        v:=m(u-1); {v = u − 1 ∧ u = x + 1}
        x:=v+1; {u = x}
    fi; {u = x}
    return x
}
```

2.5 Conclusions and related work

In this chapter we have introduced a sequential class-based object-oriented language, specified its semantics, and developed a proof system to prove safety properties of programs written in the language. The programming language allows dynamic object creation, aliasing, method invocation, and recursion.

We represent method invocations by two synchronous communication events between the caller and the callee object, one for the call and one for returning.

Thus, though the language is sequential, i.e., we don't have shared-variable concurrency between threads, we have concurrency between objects.

To support a clean interface between internal and external object behavior, we have excluded qualified references $e.x$ to instance variables. To mirror this modularity in the logic, the assertion language, used to describe program properties, consists of two levels: The local language allows to describe the execution of method instances in terms of their local variables and of the instance variables of the object to which they belong. The global language reasons about the global state and is used to describe communication properties. This two-level assertion language allows a modular annotation and verification process: The invariance of object properties, as specified by the class annotation, is independent of the definition and annotation of other classes, as long as the assumptions about the communication properties hold.

The proof system defines a number of verification conditions, which, applied to proof outlines, assure inductivity and thus invariance of their annotation. This is proved in Section 6. The above modularity is present also in the verification conditions: Local correctness and interference freedom describe intra-object execution and interleaving, and are formulated in the local language. Communication and object creation refer in general[6] to inter-object computation; the corresponding conditions of the cooperation test use the global language.

In the following we discuss related work on the semantics of sequential *Java* sublanguages in Section 2.5.1. Research results related to our proof system are handled in Section 2.5.2.

2.5.1 Semantics

Though we use an abstract syntax, the programming language can be seen as a *Java* subset. Besides the official Sun reference [GJSB00] there exists a number of introductions and references to the *Java* language, see, e.g., [Gra97].

The official Sun references for *Java* are sometimes inconsistent and incomplete; thus there is a need to develop formal models describing the behavior of *Java* programs. The size of the language makes it hard to develop a complete formal specification for it. Studies usually focus on some special aspects and abstract away other details. The book [AF99] is a collection of works in the field of *Java*'s formal syntax and semantics.

There are many research groups working on the formalization of sequential *Java* sublanguages. Drossopoulou et al. [DEK99] give a formal description of the type system and operational semantics of a sequential *Java* sublanguage with inheritance, and prove soundness of the type system. Syme [Sym97, Sym99] encodes some of the models of Drossopoulou et al. in his DECLARE system, and gives a machine-checked type-soundness proof. Drossopoulou and Valkevych [DV00] present a type system and a semantics for a *Java* subset including exception handling, where they distinguish if a thrown exception is handled (caught) or not.

[6]When the object communicated with is not identical to the value of this.

A language with inner classes and inheritance is formalized by Igarashi and Pierce [IP00]. Igarashi et al. [IPW99] develop type rules and an operational semantics for a small sequential sublanguage with subtyping (Featherweight Java) and prove type safety. Their calculus is smaller than CLASSICJAVA proposed by Flatt et al. in [FKF99].

Alves-Foss and Lam [AFL99] present a dynamic denotational semantics of a *Java* subset. The semantics covers almost the full range of the base language, but excludes concurrency.

Glesner and Zimmermann [GZ98] specify the type system for a *Java* fragment with inheritance as an example of their work on many-sorted logic.

2.5.2 Proof system

In contrast to our work, not all deductive approaches for concurrent systems use auxiliary variables[7]. Lamport [Lam88] uses *control predicates*, which are assertions explicitly mentioning the control state. Our proof system defines some *built-in* auxiliary variables which are similar to Lamport's control predicates. The built-in auxiliary variables are updated by a built-in augmentation; the user does not have to augment the program with them, but may use their values in the user-definable part of the augmentation and in the annotation. Additionally, in our approach the user may define further arbitrary auxiliary variables, according to his or her need to specify the annotation. From this point of view, one can say that our proof system combines control predicates and auxiliary variables.

Other approaches [AL97, JKW03, vON02] based on the global store model use a full semantic embedding to reason about invariant program properties. This means that assertions are predicates over configurations and not over states. Implicitly, those approaches do not require augmentation. Invariance of the annotation under execution can be shown directly using the (usually denotational) semantics.

In our approach, invariance of the annotation is assured by the verification conditions of our proof system, which are logical implications (see Chapter 5) evaluated in states. The main advantages of our syntactic approach is that we only have to encode states and the semantics of assertions in the theorem prover, since the verification conditions are implications evaluated in states. In contrast, the semantic approaches require an embedding of the programming language semantics in the theorem prover.

The grouping of the verification conditions of our proof system is standard. As already mentioned in Section 1.3, the issue of local correctness goes back to Hoare's logic [Hoa69] developed for a sequential language. Owicki and Gries [OG76] (see also [Owi75]) and Lamport [Lam77] extended the logic to shared-variable concurrency, thereby formalizing an interference freedom test, and giving the notion of a proof outline the first time. The cooperation test

[7] They are also called "dummy variables", "ghost variables", and "thought variables".

was first introduced for CSP by Apt, Francez, and de Roever [AFdR80] and by Levin and Gries [LG81].

In the field of deductive verification support for object-oriented programs, research mostly concentrated on *sequential* languages. Early examples of Hoare-style proof systems for sequential object-oriented languages are worked out by de Figueiredo [dF95] and by Leavens and Wheil [LW90, LW95].

De Boer [dB91b, dB99] develops a first sound and relatively complete proof system for a sequential object-oriented language called SPOOL. Later work [PdB03, dBP03, dBP02] includes more features, especially inheritance and subtyping.

The aim of the work in the LOOP project (Logic of Object-Oriented Programming) [Loo01] is to specify and verify properties of classes in class-based object-oriented languages. The project research concentrates on a sequential subpart of *Java*; the main focus of application is *JavaCard*.

A compiler [vdBJ02] translates programs and their specifications into *PVS* [JvdBH$^+$98, JvdBH$^+$98] and *Isabelle/HOL* [vdBHJP00]. The translation is based on the embedding of a coalgebraic semantics of a sequential *Java* subset into Higher Order Logic (HOL). Soundness of the representation is shown in [Hui01]. LOOP specifications, formalized in *JML*, are represented in HOL by a set of proof rules [JP01]. Jacobs presents also a coalgebraic view of exceptions in [Jac01]. Modeling inheritance in higher order logic is the topic of [HJ00]. The LOOP tool and its methodology have been applied to several case studies; see, e.g., [PvdBJ01, PvdBJ00, vdBJP01, HJvdB01, JKW03].

Though research within the LOOP project deals with many of the complexities of *Java*, they neither handle concurrency, nor investigate completeness.

The project Bali [Bal03] is concerned with the formalization of various aspects of *Java* in the theorem prover *Isabelle/HOL* [Pau93]. Nipkow and von Oheimb [NvO98, vON99] prove type soundness of their $Java_{light}$ subset, a large sequential sublanguage of *Java*. They formalize its abstract syntax, its type system, and well-formedness conditions, and develop an operational semantics. Based on this formalization, they express and prove type soundness within the theorem prover *Isabelle/HOL*. To complement the operational semantics of $Java_{light}$, von Oheimb presents an axiomatic semantics [vO00a, vO00b], and proves soundness and completeness of the latter with respect to the operational semantics.

With μ*Java*, Nipkow et al. [NvOP00] offer an *Isabelle/HOL* embedding of *Java*'s imperative core with classes. They present a static and a dynamic semantics of the language both at the *Java* level and the *JVM* level.

Based on [NvOP00], von Oheimb [vO01] presents a Hoare-style calculus for a *JavaCard* subset and proves soundness and completeness in *Isabelle/HOL*. Nipkow [Nip02] selects some of the technically difficult language features and deals with their Hoare logic in isolation. The combination of [vO01] and [Nip02] in one language (NanoJava) is formulated in [vON02].

In contrast to our approach, the Bali project aims to cover only sequential subsets of *Java*. Furthermore, a semantic representation of assertions is used; program execution is specified by state transformations. Our proof system uses a syntactic representation and substitution operators instead of state transformations. We see the main advantage of such a syntactical representation in an increased automation of computer-supported verification: Using a theorem prover to prove program properties correct requires only the representation of the assertion semantics in the theorem prover, but not the programming language semantics as in more semantically-oriented approaches. Our experience shows that this simple representation leads to a high degree of automation. A disadvantage inherent to the syntactical approach is that it does not support a computer-assisted soundness proof.

Poetzsch-Heffter and Müller [PH97a, PH97b, PHM98, PHM99] develop a Hoare-style programming logic for a sequential kernel of *Java*, featuring interfaces, subtyping, and inheritance. Translating the operational and the axiomatic semantics into the HOL theorem prover allows a computer-assisted soundness proof. Neither this group deals with concurrent sublanguages of *Java*.

Reus, Wirsing, and Hennicker [RW00, RHW01] use a modification of the *object constraint language* OCL (OCL^{light}) as assertional language to annotate UML class diagrams and to generate proof conditions for *Java*-programs. They treat inheritance and show soundness of the proof system.

Abadi and Leino [AL97] present a Hoare-style proof-system for a sequential object-oriented language in the form of an object calculus [AC96]. They also prove soundness of their logic. Their language features heap-allocated objects (but no classes), side-effects and aliasing, and its type system supports subtyping. Their assertion language is presented as an extension of the object calculus' language of type and analogously, the proof system extends the type derivation system. The close connection of types and specifications in the presentation is exploited by Tang and Hofmann in [TH02] for the generation of verification conditions.

The aim of the KeY project [KeY03] is to integrate formal software specification and verification into the industrial software engineering process [ABB+00]. The starting point is a commercial CASE tool which will be augmented by capabilities for formal specification and verification. The paper [Bec01] describes a dynamic logic for *JavaCard* and a sequent calculus for this logic, which is the basis for the KeY system's software verification component. The research is guided and evaluated through an extended case study using *JavaCard* applets as an application domain. A case study can be found in [BH03].

Chapter 3

The concurrent language

In this chapter we extend the language $Java_{seq}$ to a *concurrent* language $Java_{conc}$ by allowing *dynamic thread creation*. Again, we define syntax and semantics of the language in the Sections 3.1 and 3.2, before formalizing the proof system for the concurrent language in Section 3.3. Section 3.4 contains concluding remarks and discusses related work.

3.1 Syntax

Expressions, statements, and methods can be constructed as in $Java_{seq}$. The abstract syntax of the remaining constructs is summarized in Table 3.1. As

$$
\begin{array}{rcl}
\mathit{class} & ::= & \mathsf{class}\ c\{\mathit{meth}\ldots\mathit{meth}\ \mathit{meth}_{\mathsf{run}}\ \mathit{meth}_{\mathsf{start}}\} \\
\mathit{class}_{\mathsf{main}} & ::= & \mathit{class} \\
\mathit{prog} & ::= & \mathit{class}\ldots\mathit{class}\ \mathit{class}_{\mathsf{main}}
\end{array}
$$

Table 3.1: $Java_{conc}$ abstract syntax

we focus on concurrency aspects, all classes are `Thread` classes in the sense of *Java*: Each class contains a predefined `start` method that can be invoked only once for each object, resulting in a new thread of execution. The new thread starts to execute the user-defined `run` method of the given object while the initiating thread continues its own execution. The `run` methods cannot be invoked directly. The parameterless `start` method without return value is not implemented syntactically; see the next section for its semantics. Note, that the syntax does not allow qualified references to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object only, but not across object boundaries.

3.2 Semantics

The operational semantics of $Java_{conc}$ extends the semantics of $Java_{seq}$ by dynamic thread creation. The additional rules are shown in Table 3.2. The invo-

$$\frac{\beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in Val^c(\sigma) \qquad \neg started(T \cup \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \beta)}{\begin{array}{l}\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, stm), (\beta, \tau_{init}^{\mathsf{run},c}, body_{\mathsf{run},c})\}, \sigma\rangle\end{array}} \ \textsc{Call}_{start}$$

$$\frac{\beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in Val(\sigma) \qquad started(T \cup \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \beta)}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle} \ \textsc{Call}_{start}^{skip}$$

Table 3.2: $Java_{conc}$ operational semantics

cation of a start method brings a new thread into being (rule $\textsc{Call}_{start}$) which starts to execute the run method of the callee object[1]. Only the first invocation of the start method has this effect (rule $\textsc{Call}_{start}^{skip}$).[2] This is captured by the predicate $started(T, \beta)$ which holds iff there exists a stack of the form $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n) \in T$ such that $\beta = \alpha_0$. A thread ends its lifespan by returning from a run method (rule $\textsc{Return}_{run}$ of Table 2.3).[3]

3.3 The proof system

In contrast to the sequential language, the proof system additionally has to accommodate dynamic thread creation and shared-variable concurrency. From a proof theoretical view, the latter is the main difference with respect to the sequential case. Before describing the proof method, we show how to extend the built-in augmentation of the sequential language.

[1] We define thread creation to be atomic. In *Java*, however, first the predefined start method is called. During the execution of the start method a new thread gets created, which finally invokes the callee's run method. Note that run methods must not be synchronized, and that during thread creation no instance states get modified. Consequently, though *Java* defines a finer-grained semantics allowing additional control points, reachability for the remaining, common control points is identical in both semantics.

[2] In *Java* an exception is thrown if the thread is already started but not yet terminated.

[3] The worked-off local configuration (α, τ, ϵ) is kept in the global configuration to ensure that the thread of α cannot be started twice.

3.3.1 Proof outlines

To obtain a complete proof system, for the concurrent language we additionally have to be able to identify *threads*. We identify a thread by the object in which it has begun its execution. We use the type Thread thus as abbreviation for the type Object. This identification is unique, since an object's thread can be started only once. During a method call, the callee thread receives its own identity as a built-in auxiliary formal parameter thread. Additionally, we extend the auxiliary formal parameter caller by the caller thread identity, i.e., let caller be of type $\mathsf{Object} \times \mathsf{Int} \times \mathsf{Thread}$, storing the identities of the caller object, the calling local configuration, and the caller thread. Note that the thread identities of caller and callee are the same in all cases except for the invocation of a start method. We need this additional information identifying the caller thread, because in a self-invocation of the start method the corresponding observation of the caller thread must not change the instance state (see page 29); thus the only way to refer to the caller thread in an observation of the call is to make this identity known by the new thread, which may execute observations modifying the instance state. The run method of the initial object is executed with the parameters $(\mathsf{thread}, \mathsf{caller})$ having the values $(\alpha_0, (null, 0, null))$, where α_0 denotes the initial object. The value of the boolean built-in auxiliary instance variable $\mathsf{started}$, finally, remembers whether the object's start method has already been invoked.

Syntactically, each formal parameter list $\vec{u}$ in the original program gets extended to $(\vec{u}, \mathsf{thread}, \mathsf{caller})$. Correspondingly for the caller, each actual parameter list $\vec{e}$ in statements invoking a method different from start gets extended to $(\vec{e}, \mathsf{thread}, (\mathsf{this}, \mathsf{conf}, \mathsf{thread}))$. The invocation of the parameterless start method of an object e_0 gets the actual parameter list $(e_0, (\mathsf{this}, \mathsf{conf}, \mathsf{thread}))$. Finally, the callee observation at the beginning of the run method executes $\mathsf{started} := \mathsf{true}$. The variables conf and $\mathsf{counter}$ are updated as in the previous chapter. Again, for a detailed description of the syntactical built-in augmentation we refer to Section 9.2.2.

Remember that the caller observation of self-calls may not modify the instance state, as required in Section 2.4.1. Invoking the start method by a self-call is specific in that, when the thread is already started, the caller is the only active entity. In this case, it has to be the caller that updates the instance state; the corresponding observation has the form $x := \mathsf{if}\ e_0 = \mathsf{this} \wedge \neg \mathsf{started}\ \mathsf{then}\ x\ \mathsf{else}\ e\ \mathsf{fi}$.

Since a thread calling a start method does not wait for return but continues execution, the augmentation and annotation of such method invocations have the form $\{p_1\}\, e_0.\mathsf{start}(\vec{e})\ \{p_2\}^{!call} \langle stm \rangle^{!call} \{p_3\}$.

3.3.2 Verification conditions

Initial correctness

Initial correctness changes only in that the formal parameters thread and caller get assigned the initial values α and $(null, 0, null)$, where α is the initial object. We modify the initial correctness conditions of the previous chapter (page 33) correspondingly as follows:

Definition 3.3.1 (Initial correctness) *Let the body of the* run *method of the main class c be* $\{p_2\}^{?call}\langle\vec{y}_2 := \vec{e}_2\rangle^{?call}\{p_3\}$ *stm*; return *with local variables $\vec{v}$ without the formal parameters, $z \in LVar^c$, and $z' \in LVar^{\mathsf{Object}}$. A proof outline is* initially correct, *if*

$$\models_{\mathcal{G}} \quad \{\mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'\} \quad (3.1)$$
$$\vec{v}, \mathsf{thread}, \mathsf{caller} := \mathsf{Init}(\vec{v}), z, (\mathsf{null}, 0, \mathsf{null})$$
$$\{P_2(z)\}\ ,\ \textit{and}$$
$$\models_{\mathcal{G}} \quad \{\mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'\} \quad (3.2)$$
$$\vec{v}, \mathsf{thread}, \mathsf{caller} := \mathsf{Init}(\vec{v}), z, (\mathsf{null}, 0, \mathsf{null});\quad z.\vec{y}_2 := \vec{E}_2(z)$$
$$\{GI \wedge P_3(z) \wedge I_c(z)\}\ .$$

Again, the assertion $\mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'$ states that the initial global state defines exactly one existing object z being in its initial instance state, and the observation $\vec{y}_2 := \vec{e}_2$ at the beginning of the run method of the initial object z is represented by the assignment $z.\vec{y}_2 := \vec{E}_2(z)$. The difference is in the initialization of the local configuration, which is now represented by the assignment $\vec{v}, \mathsf{thread}, \mathsf{caller} := \mathsf{Init}(\vec{v}), \mathsf{thread}, (\mathsf{null}, 0, \mathsf{null})$.

Local correctness

Local correctness is not influenced by the new issue of concurrency. Note that local correctness applies now to all concurrently executing threads.

The interference freedom test

Interference of a *single* thread under its own execution remains the same as for the sequential language. However, we additionally have to deal with invariance of properties of a thread under the execution of a *different* thread. Note that assertions at auxiliary points do not have to be shown invariant.

An assertion q at a control point has to be invariant under an assignment $\vec{y} := \vec{e}$ in the same class only if the local configuration described by the assertion is not active in the computation step executing the assignment. Again, to distinguish local variables of the different local configurations, we rename those of the assertion which has to be shown invariant, resulting in primed variables, expressions, and assertions. For example, in the conditions we use thread to identify the thread executing the assignment, and thread′ to identify the thread described by q.

If q and $\vec{y} := \vec{e}$ belong to the *same* thread, i.e., $\mathsf{thread}' = \mathsf{thread}$, then we have the same antecedent as for the sequential language. If the assertion and the assignment belong to *different* threads, interference freedom must be shown in all cases except for the self-invocation of the start method: The callee observation of a self-invocation of a start method cannot interfere with the precondition of the invocation. To describe this setting, we define $\mathsf{self\_start}(q, \vec{y} := \vec{e})$ by $\mathsf{caller} = (\mathsf{this}, \mathsf{conf}', \mathsf{thread}')$ iff q is the precondition of a method invocation

e_0.start($\vec{e}$) and the assignment is the callee observation at the beginning of the run method, and by false, otherwise.

The example of Figure 3.1 illustrates the execution of an assignment in an object, in which two threads are executing concurrently, sharing the instance variables of the object. The assignment occurs in a method which was invoked by a self-call; p_7 describes the control point in the caller configuration, and p_8 and p_9 are the pre- and postconditions of the assignment. The other thread has currently two control points in the object: p_4 describes the local configuration on the top of the stack, and p_2 is at a control point waiting for return. All other control points are in objects different from the one in which the assignment is executed.

Both p_2 and p_4, describing a thread different from the executing one, have to be invariant under the assignment. Also p_7 has to be invariant if the assignment does not observe return. The assertion p_8 does not have to be invariant, where satisfaction of p_9 after execution is assured by local correctness. We do not have to show invariance of p_1, p_3, p_5, and p_6, since these assertions describe objects different from the one in which the assignment is executed.

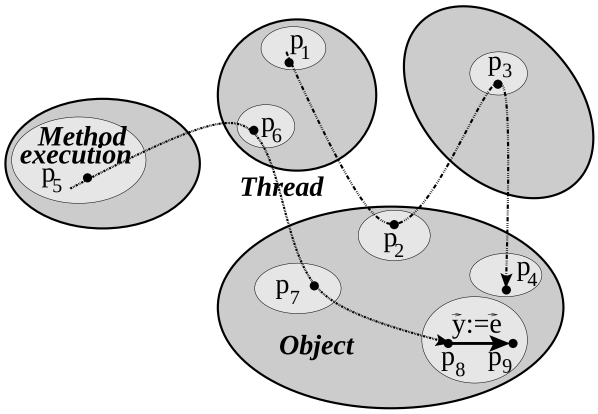


Figure 3.1: Interference between threads

Definition 3.3.2 (Interference freedom) *A proof outline is* interference free, *if the conditions of Definition 2.4.11 hold with* waits_for_ret$(q, \vec{y} := \vec{e})$ *replaced by*

$$\begin{aligned}\mathsf{interferes}(q, \vec{y} := \vec{e}) \stackrel{def}{=} \;& \mathsf{thread} = \mathsf{thread}' \rightarrow \mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e}) \wedge \\ & \mathsf{thread} \neq \mathsf{thread}' \rightarrow \neg\mathsf{self\_start}(q, \vec{y} := \vec{e})\,.\end{aligned}$$

Example 3.3.3 *Assume an annotated assignment* $\{p\}$ *stm in a method, and an assertion* q *at a control point not waiting for return in the same method, such that both* p *and* q *imply* thread = this. *I.e., the method is executed only* *by the thread of the object to which it belongs. Clearly, p and q cannot be simultaneously reached by the same thread. For invariance of q under the assignment stm, the antecedent of the interference freedom condition implies $p \wedge q' \wedge \mathsf{interferes}(q, stm)$. From $p \wedge q'$ we conclude $\mathsf{thread} = \mathsf{thread}'$, and thus by the definition of $\mathsf{interferes}(q, stm)$ the assertion q should be at a control point waiting for return, which is not the case, and thus the antecedent of the condition evaluates to false.*

Example 3.3.4 *Consider the following method which increments the value of an instance variable x:*

```
inc(){ x:=x+1 }
```

Under which conditions can we prove invariance of the specification

inc(){ $\{x = 0\}$ x:=x+1 $\{x = 1\}$ }

under the assignment $x := x + 1$?

One possible condition is that only the thread originating from the object itself can execute this method:

inc(){ $\{x = 0 \wedge \mathsf{thread} = \mathsf{this}\}$ x:=x+1 $\{x = 1\}$ }

We have the following interference freedom condition for invariance of the precondition of the assignment under the execution of the assignment:

$$\models_{\mathcal{L}} (x = 0 \wedge \mathsf{thread} = \mathsf{this}) \wedge (x = 0 \wedge \mathsf{thread}' = \mathsf{this}) \wedge \mathsf{thread} \neq \mathsf{thread}' \rightarrow x = 0\,.$$

Note that since the assertion $x = 0 \wedge \mathsf{thread} = \mathsf{this}$ is not at a control point waiting for return, the definition of interferes *assures* $\mathsf{thread} \neq \mathsf{thread}'$.

The specification would also be invariant under the weaker requirement that though also other threads may execute m*, but not concurrently. This property we can express using an auxiliary instance variable t storing the identity of the thread executing* m*:*

inc(){ $\{t = \mathsf{null} \wedge \mathsf{thread} \neq \mathsf{null}\}^{?call}$ $\langle t := \mathsf{thread}\rangle^{?call}$
    $\{x = 0 \wedge \mathsf{thread} = t \neq \mathsf{null}\}$ x:=x+1; $\{x = 1 \wedge \mathsf{thread} = t \neq \mathsf{null}\}$
    return $\{\mathsf{thread} = t \neq \mathsf{null}\}^{!ret}$ $\langle t := \mathsf{null}\rangle^{!ret}$
}

As a last example, we could also state that the method is not executed concurrently, by storing the identity of the executing configuration in t:

inc(){ $\{t = -1\}^{?call}$ $\langle t := \mathsf{conf};\rangle^{?call}$
    $\{x = 0 \wedge \mathsf{conf} = t \geq 0\}$ x:=x+1; $\{x = 1 \wedge \mathsf{conf} = t \geq 0\}$
    return $\{\mathsf{conf} = t \geq 0\}^{!ret}$ $\langle t := -1\rangle^{!ret}$
}

where the class invariant states $\mathsf{counter} \geq 0$.

Note that in the latter both cases the preconditions of the observation of the call and of the return,as well as the precondition of the assignment are justified in the cooperation test.

The cooperation test

The cooperation test for object creation is not influenced by adding concurrency. Also the invocation of methods different from start, executed by a single thread, is not affected by the presence of concurrency. However, we have to extend the cooperation test for communication by defining additional conditions for thread creation. In the definition below, the first case (CALL_{start}) covers the creation of a new thread by invoking a start method. Again, z and z' are fresh logical variables representing the caller and the callee object. Besides the precondition of the call, the global, and the class invariants, we assume that the execution is enabled, i.e., that the thread of the callee object is not yet started, as expressed by $\neg z'.\mathsf{started}$. Invoking the start method of an object whose thread is already started does not have communication effects ($\text{CALL}^{skip}_{start}$). The same holds for returning from a run method, which is already included in the conditions for the sequential language as for the termination of the only thread (case RETURN_{run} on page 42). Note that this condition applies now to all threads.

Definition 3.3.5 (Cooperation test: Communication) *A proof outline satisfies the* cooperation test for communication, *if the conditions of Definition 2.4.14 hold for the statements listed there with $m \neq \mathsf{start}$, and additionally in the following cases:*

1. CALL_{start}*: For all statements $\{p_1\}\, e_0.\mathsf{start}(\vec{e})\ \{p_2\}^{!call}\langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_3\}$ in class c with e_0 of type c',* comm *is given by $E_0(z) = z' \wedge \neg z'.\mathsf{started}$, where $\{q_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\, stm;\mathsf{return}$ is the body of the* run *method of c' having formal parameters $\vec{u}$, and local variables $\vec{v}$ except the formal parameters. The callee class invariant is $q_1 = I_{c'}$. Furthermore, f_{comm} is $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$, f_{obs1} is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{obs2} is $z'.\vec{y}'_2 := \vec{E}'_2(z')$.*

2. $\text{CALL}^{skip}_{start}$*: For the above statements, the equations must additionally hold with the assertion* comm *given by $E_0(z) = z' \wedge z'.\mathsf{started}$, $q_2 = q_3 = \mathsf{true}$, q_1 and f_{obs1} as above, and f_{comm} and f_{obs2} are the empty statement.*

Examples

Example 3.3.6 *Assume the following augmented and annotated class with integer instance variables* `thr`*,* `nr`*, and* `sum`*, an auxiliary integer instance variable* `between`*, and where its class invariant I is given by* `sum=(thr-between)*nr`*:*

```
class Sum{
    Int thr,nr,sum;
    ⟨Int between;⟩
    {sum = (thr − between) ∗ nr}

    Void inc(){
        thr := thr+1;   ⟨between := between + 1⟩^ass
        sum := sum+nr   ⟨between := between − 1⟩^ass
    }
}
```

Each thread that executes the method `inc` *increases the value of the instance variable* `thr` *by one and the value of the instance variable* `sum` *by the constant value* `nr`*. This way,* `thr` *stores the number of invocations of* `inc` *and if no threads are in the* `inc` *method then* `sum` *equals* `thr*nr`*, as expressed by the annotation.*

There are no local correctness and no cooperation test conditions. Two interference freedom conditions assure the invariance of the class invariant under the assignments in the `inc` *method. Both conditions are verified automatically in PVS.*

Example 3.3.7 *Assume the following proof outline which offers mutual exclusion for the execution of a critical section within the method* `mutex`*:*

```
class MutEx extends Thread{
    Int t;
    ⟨listThread tseq;⟩
    ⟨Thread crit;⟩

    {t = |tseq|}

    Void mutex(){
        Bool done;

        {thread ≠ null ∧ thread ≠ crit}
        done := false;
        {done ∨ (thread ≠ null ∧ thread ≠ crit)}
        while (¬done) do
            {thread ≠ null ∧ thread ≠ crit}
            t := t+1; ⟨tseq := tseq ∘ thread⟩^ass
            {thread ≠ null ∧ thread ≠ crit ∧ thread ∈ tseq}
            if (t>1) ⟨crit := (if t > 1 then crit else thread fi)⟩ then
                {thread ≠ null ∧ thread ≠ crit ∧ thread ∈ tseq}
                t:=t-1; ⟨tseq := tseq − thread⟩^ass
                {thread ≠ null ∧ thread ≠ crit}
            else
                {thread ≠ null ∧ thread = crit ∧ thread ∈ tseq}
                //critical section
                done:=true;
                {thread ≠ null ∧ thread = crit ∧ thread ∈ tseq ∧ done}
                t:=t-1; ⟨crit, tseq := null, tseq − thread⟩^ass
                {done}
            fi
            {done ∨ (thread ≠ null ∧ thread ≠ crit)}
        od
    }
}
```

The annotation `thread=crit` *for the critical section expresses that there is at most one thread executing the critical section, whose identity is stored in the auxiliary instance variable* `crit`*. The identities of the threads which has increased but not yet decreased the value of* `t` *are stored in the auxiliary instance variable* `tseq`*.*

The Verger *tool generates*[4] 11 *local and* 29 *interference freedom conditions*[5]*. Cooperation test conditions are not generated. All conditions are verified in PVS.*

[4] for an equivalent program in Java syntax

[5] The tool does not generate trivial conditions, like for example the invariance of an assertion under an assignment to a variable which does not occur in the assertion.

3.4 Conclusions and related work

This chapter extends the previous one by adding concurrency. After describing syntax and semantics of the concurrent language, we discussed how to extend the proof system to cover multithreading.

The rules of the operational semantics of the sequential language are extended by rules for dynamic thread creation; the transition rules for the sequential language are not modified. Also the verification conditions are extended, i.e., the conditions of the sequential language are not modified, we just define additional conditions to cover concurrency. This fact reflects the one-to-one connection between the transition rules of the semantics and the verification conditions of the proof system.

In his book, Lea [Lea99] gives a general introduction to concurrency in *Java*. Other introductory books on *Java* multithreading are, e.g., [OW99, CT00, Hol00, Hyd01, LB99]. Magee and Kramer offer in [MK99] an approach for designing, analyzing and implementing concurrent programs.

First we collect related work on the semantics of multithreaded *Java* sublanguages, before discussing proof systems for such languages.

3.4.1 Semantics

Börger et al. presented several results on formal specifications of *Java*, the *JVM*, and the compiler. Their work is based on the Abstract State Machine (ASM) formalism [BS03]. Two earlier papers specify a modular semantics of a subset of the *JVM* [BS99a] and a subset of *Java* [BS99b]. In [BS98] they state correctness of the compiler for parts of these subsets. In [BS00] these authors discuss the exception handling mechanism, and formulate the correctness of compiling exception handling with a full proof.

The book [SSB01] by Stärk, Schmid, and Börger provides a formal specification of *Java* and of the *JVM*, developed incrementally in different layers. The work includes a compiler of *Java* programs to *JVM* code and a bytecode verifier. Correctness of the compiler with respect to the given semantics of *Java* and the *JVM*, and its completeness with respect to the bytecode verifier are formally proven (see also [SS03]).

Gurevich, Schulte, and Wallace [Wal97, GSW00a, GSW00b] give the specification of a multithreaded *Java* subset with exception handling. Their work is also based on the Abstract State Machine framework.

3.4.2 Proof system

Work on *proof systems* for parallel object-oriented languages in general and for multithreading aspects of *Java* in particular is rather scarce.

America and de Boer [AdB90a, AdB93] develop proof systems for a language with dynamic process creation. De Boer [dB99, dB91b, dB91a, dB90] presents a sound and relatively complete proof system in weakest precondition formulation for a parallel object-based language called POOL, i.e., without inheritance and subtyping, and also without reentrant method calls. POOL has a different object-oriented concurrency model where each object specifies its own thread of control.

While our proof systems takes full concurrency into account, other research directions try to reduce the complexity by putting constraints on programs. Flanagan, Freund, and Qadeer describe in [FFQ02] a static checker for multithreaded software systems. The programmer should specify environment assumptions that put constraints onto the interaction between threads. The checker uses this information to reduce the verification of the original multithreaded program to the verification of several sequential programs.

Chapter 4

Reentrant monitors

In this chapter we extend the concurrent language with *monitor synchronization*. Again, we define syntax and semantics of the language $Java_{synch}$ in the Sections 4.1 and 4.2 before formalizing the proof system in Section 4.3. We conclude in Section 4.4 with some remarks and related work.

As a mechanism of concurrency control, methods can be declared as *synchronized.* Each object has a *lock* which can be owned by at most one thread. Synchronized methods of an object can be invoked only by a thread that owns the lock of that object. If the thread does not own the lock, it has to wait until the lock gets free. A thread owning the lock of an object can recursively invoke several synchronized methods of that object; this corresponds to the notion of reentrant monitors.

Besides mutual exclusion, using the lock-mechanism for synchronized methods, objects offer the methods wait, notify, and notifyAll as means to facilitate thread coordination at the object boundary. A thread owning the lock of an object can block itself and free the lock by invoking wait on the given object. The blocked thread can be reactivated by another thread owning the lock via the object's notify method; the reactivated thread must reapply for the lock before it may continue its execution. The method notifyAll, finally, generalizes notify in that it notifies all threads blocked on the object.

4.1 Syntax

Expressions and statements can be constructed as in the previous languages. The abstract syntax of the remaining constructs is summarized in Table 4.1.

Methods are decorated by a modifier *modif* distinguishing between *non-synchronized* and *synchronized* methods.[1] In the sequel we also refer to statements in the body of a synchronized method as being synchronized. Furthermore, we consider the additional predefined methods wait, notify, and notifyAll,

[1] *Java* does not have the "non-synchronized" modifier: methods are non-synchronized by default.

$$
\begin{array}{rcl}
\mathit{modif} & ::= & \mathsf{nsync} \mid \mathsf{sync} \\
\mathit{meth} & ::= & \mathit{modif}\, m(u,\ldots,u)\{\ \mathit{stm}; \mathsf{return}\ \mathit{exp}_{ret}\} \\
\mathit{meth}_{\mathsf{run}} & ::= & \mathsf{nsync\ run}()\{\ \mathit{stm}; \mathsf{return}\ \} \\
\mathit{meth}_{\mathsf{wait}} & ::= & \mathsf{nsync\, wait}()\{\ ?\mathsf{signal}; \mathsf{return}_{getlock}\ \} \\
\mathit{meth}_{\mathsf{notify}} & ::= & \mathsf{nsync\, notify}()\{\ !\mathsf{signal}\,; \mathsf{return}\ \} \\
\mathit{meth}_{\mathsf{notifyAll}} & ::= & \mathsf{nsync\, notifyAll}()\{\ !\mathsf{signal\_all}; \mathsf{return}\ \} \\
\mathit{meth}_{predef} & ::= & \mathit{meth}_{\mathsf{start}}\ \mathit{meth}_{\mathsf{wait}}\ \mathit{meth}_{\mathsf{notify}}\ \mathit{meth}_{\mathsf{notifyAll}} \\
\mathit{class} & ::= & \mathsf{class}\ c\{\mathit{meth}\ldots\mathit{meth}\ \mathit{meth}_{\mathsf{run}}\ \mathit{meth}_{predef}\} \\
\mathit{class}_{\mathsf{main}} & ::= & \mathit{class} \\
\mathit{prog} & ::= & \mathit{class}\ldots\mathit{class}\ \mathit{class}_{\mathsf{main}}
\end{array}
$$

Table 4.1: Java_{synch} abstract syntax

whose definitions use the auxiliary statements !signal, !signal_all, ?signal, and $\mathsf{return}_{getlock}$, which describe at a high level of abstraction the signal-and-continue mechanism underlying the wait, notify, and notifyAll methods.[2]

4.2 Semantics

The operational semantics extends the semantics of Java_{conc} by the rules of Table 4.2, where the CALL rule is replaced. For synchronized method calls, the lock of the callee object has to be free or owned by the executing thread, as expressed by the predicate *owns*, defined below.

The remaining rules handle the semantics of the monitor methods wait, notify, and notifyAll. In all three cases the caller must own the lock of the callee object (rule $\text{CALL}_{monitor}$). A thread can block itself on an object whose lock it owns by invoking the object's wait method, thereby relinquishing the lock and placing itself into the object's wait set. Formally, the wait set $\mathit{wait}(T,\alpha)$ of an object is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, ?\mathsf{signal}; \mathit{stm})$. After having put itself on ice, the thread awaits notification by another thread which invokes the notify method of the object. The !signal-statement in the notify method thus reactivates a non-deterministically chosen single thread waiting for notification on the given object (rule SIGNAL). Analogously to the wait set, the notified set $\mathit{notified}(T,\alpha)$ of α denotes the set of all stacks in T with top element of the form $(\alpha, \tau, \mathsf{return}_{getlock})$, i.e., threads which have been notified and trying to get hold of the lock again. According to rule RETURN_{wait}, the receiver can continue after notification in executing $\mathsf{return}_{getlock}$ only if the lock is free. Note that the notifier does not hand over the lock to the one being notified but continues to own it. This behavior is known as the *signal-and-continue* monitor discipline [And00] (cf.

[2] *Java*'s `Thread` class additionally supports methods for suspending, resuming, and stopping a thread, but they are deprecated and thus not considered here.

$$\frac{\begin{array}{c} m \notin \{\mathsf{start}, \mathsf{run}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\} \qquad \mathit{modif}\, m(\vec{u})\{\ \mathit{body}\ \} \in \mathit{Meth}_c \\ \beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in \mathit{Val}^c(\sigma) \qquad \tau' = \tau_{\mathit{init}}^{m,c}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \\ (\mathit{modif} = \mathsf{sync}) \rightarrow \neg \mathit{owns}(T,\beta) \end{array}}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,u := e_0.m(\vec{e}); \mathit{stm})\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,\mathsf{receive}\, u; \mathit{stm}) \circ (\beta,\tau',\mathit{body})\}, \sigma\rangle \end{array}} \ \textsc{Call}$$

$$\frac{\begin{array}{c} m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\} \\ \beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in \mathit{Val}^c(\sigma) \qquad \mathit{owns}(\xi \circ (\alpha,\tau,e.m(); \mathit{stm}), \beta) \end{array}}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,e.m(); \mathit{stm})\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,\mathsf{receive}; \mathit{stm}) \circ (\beta,\tau_{\mathit{init}}^{m,c},\mathit{body}_{m,c})\}, \sigma\rangle \end{array}} \ \textsc{Call}_{\mathit{monitor}}$$

$$\frac{\neg \mathit{owns}(T,\beta)}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,\mathsf{receive}; \mathit{stm}) \circ (\beta,\tau',\mathsf{return}_{\mathit{getlock}})\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,\mathit{stm})\}, \sigma\rangle \end{array}} \ \textsc{Return}_{\mathit{wait}}$$

$$\frac{}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,!\mathsf{signal}; \mathit{stm})\} \,\dot\cup\, \{\xi' \circ (\alpha,\tau',?\mathsf{signal}; \mathit{stm}')\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,\mathit{stm})\} \,\dot\cup\, \{\xi' \circ (\alpha,\tau',\mathit{stm}')\}, \sigma\rangle \end{array}} \ \textsc{Signal}$$

$$\frac{\mathit{wait}(T,\alpha) = \emptyset}{\langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,!\mathsf{signal}; \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,\mathit{stm})\}, \sigma\rangle} \ \textsc{Signal}_{\mathit{skip}}$$

$$\frac{T' = \mathit{signal}(T,\alpha)}{\langle T \,\dot\cup\, \{\xi \circ (\alpha,\tau,!\mathsf{signal\_all}; \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T' \,\dot\cup\, \{\xi \circ (\alpha,\tau,\mathit{stm})\}, \sigma\rangle} \ \textsc{SignalAll}$$

Table 4.2: Java$_{synch}$ Operational semantics

Section 1.2). If no threads are waiting on the object, the !signal of the notifier is without effect (rule $\textsc{Signal}_{skip}$). The notifyAll method generalizes notify in that all waiting threads are notified via the !signal_all-broadcast (rule SignalAll). The effect of this statement is given by defining $signal(T, \alpha)$ as $(T \setminus wait(T, \alpha)) \cup \{\xi \circ (\alpha, \tau, stm) \mid \xi \circ (\alpha, \tau, ?\mathsf{signal}; stm) \in wait(T, \alpha)\}$.

Using the wait and notified sets, we can now formalize the *owns* predicate: A thread ξ owns the lock of β iff ξ executes some synchronized method of β, but not its wait method. Formally, $owns(T, \beta)$ is true iff there exists a thread $\xi \in T$ and a $(\beta, \tau, stm) \in \xi$ with *stm* synchronized and $\xi \notin wait(T, \beta) \cup notified(T, \beta)$. The definition is used analogously for single threads. An invariant of the semantics is that at most one thread can own the lock of an object at a time.

4.3 The proof system

The proof system has additionally to accommodate synchronization and reentrant monitors. First we define how to extend the augmentation of $Java_{conc}$, before we describe the proof method.

4.3.1 Proof outlines

To capture mutual exclusion and the monitor discipline, the built-in auxiliary instance variable lock of type Thread $\times$ Int stores the identity of the thread who owns the lock, if any, together with the number of synchronized calls in its call chain. The initial lock value $free = (null, 0)$ indicates that the lock is free. The instance variables wait and notified of type list(Thread $\times$ Int) are the analogues of the *wait* and *notified* sets of the semantics and store the threads waiting at the monitor, respectively, those having been notified. Besides the thread identity, the number of synchronized calls is stored. In other words, these variables remember the old lock-value prior to suspension which is restored when the thread becomes active again. Since the order of these sequences does not play a role, in the following we handle them as sets, and apply set-theoretical operations to them. All auxiliary variables are initialized as usual. For values *thread* of type Thread and *wait* of type list(Thread $\times$ Int), we will also write $thread \in wait$ instead of $(thread, n) \in wait$ for some n.

Syntactically, besides the built-in augmentation of the previous chapter, the callee observation at the beginning and at the end of each synchronized method body executes lock := inc(lock) and lock := dec(lock), respectively. The semantics of incrementing the lock $[\![\mathsf{inc}(\mathsf{lock})]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}$ is $(\tau(\mathsf{thread}), n+1)$ for $\sigma_{inst}(\mathsf{lock}) = (v, n)$. Note that the identity of the lock owner is set to the identity of the executing thread not only in case the lock is free, but also if a thread is already owning the lock. However, since these updates are executed in synchronized methods, the semantics assures for the latter case that the lock owner is the executing thread, i.e., if the lock is not free then the thread component of the lock is not modified. That means, incrementing the lock value (α, n) yields $(\alpha, n+1)$, whereas incrementing a free lock $(null, 0)$ by a thread

α results in $(\alpha, 1)$. Decrementing $\mathsf{dec}(\mathsf{lock})$ is done inversely: $[\![\mathsf{dec}(\mathsf{lock})]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}$ with $\sigma_{inst}(\mathsf{lock}) = (\alpha, n)$ is $(\alpha, n-1)$ if $n > 1$, and *free*, otherwise.

Instead of the auxiliary statements of the semantics, notification is represented in the state-based proof system by auxiliary assignments operating on the wait and notified variables: The auxiliary !signal and !signal_all statements are replaced by auxiliary assignments[3]. The auxiliary ?signal statements are not represented. That means, notification is represented by a single auxiliary assignment executed by the notifier. For threads being notified, the control points before and after notification are described by a single assertion in the wait method. The different control points can be distinguished by the values of the built-in auxiliary variables wait and notified.

Representing the auxiliary statements of notification by auxiliary assignments has two main advantages: First, we do not have to define verification conditions for communicating pairs of local configurations, but we can cover notification by local correctness conditions for the notifier and by interference freedom conditions for the notified partner. Second, we do not need to define special interference freedom conditions for notification. Instead, notification, being represented by auxiliary assignments, can be handled as usual assignments.

Syntactically, entering the wait method gets the observation $\mathsf{wait}, \mathsf{lock} := \mathsf{wait} \cup \{\mathsf{lock}\}, \mathsf{free}$; returning from the wait method observes $\mathsf{lock}, \mathsf{notified} := \mathsf{get}(\mathsf{notified}, \mathsf{thread}), \mathsf{notified} \backslash \{\mathsf{get}(\mathsf{notified}, \mathsf{thread})\}$. For a thread $\alpha \in Val^{\mathsf{Thread}}$ and a list $notified \in Val^{\mathsf{list}(\mathsf{Thread}\times\mathsf{Int})}$, $get(notified, \alpha)$ retrieves the value (α, n) from the list. The semantics assures uniqueness of the association. The !signal-statement of the notify method is represented by the auxiliary multiple assignment $\mathsf{wait}, \mathsf{notified} := \mathsf{notify}(\mathsf{wait}, \mathsf{notified})$, where the value $notify(wait, notified)$ is the pair of the given sets with one element, chosen nondeterministically, moved from the wait into the notified set; if the wait set is empty, it is the identity function[4]. Finally, the !signal_all-statement of the notifyAll method is represented by the auxiliary assignment $\mathsf{notified}, \mathsf{wait} := \mathsf{notified} \cup \mathsf{wait}, \emptyset$. See Section 9.2.2 for a detailed description of the syntactical augmentation.

4.3.2 Verification conditions

Initial and local correctness agree with those for $Java_{conc}$. For local correctness, note that the conditions now additionally cover invariance for threads executing notification. However, we do not need additional conditions for this case, as the effect of notification is captured by an auxiliary assignment. For threads being notified, the control points before and after notification are described by a single assertion. The interference freedom test assures invariance of this

[3]In *Java*, the implementation of the monitor methods are syntactically not included in class definitions. Their augmentation and annotation can be specified by special comments, see Section 9.2.3.

[4]Though the function *notify* is non-deterministic, it is represented in the implementation by a deterministic function, where the logic is extended by an axiom stating the properties of *notify*.

assertion under the assignment of the notifier, such that neither for this case are additional local conditions necessary.

The interference freedom test

Synchronized methods of a single object can be executed concurrently only if one of the corresponding local configurations is waiting for return: If the executing threads are different, then one of the threads executes in the non-synchronized wait method of the object; otherwise, both executing local configurations are in the same call chain. Thus we assume that either the assignment or the assertion occur outside of synchronized methods, or the assertion is at a control point waiting for return.[5]

Definition 4.3.1 (Interference freedom) *A proof outline is* interference free, *if the conditions of Definition 3.3.2 hold for all classes c, all multiple assignments $\vec{y} := \vec{e}$ with precondition p in c, and all assertions q at control points in c, such that either not both p and q occur in a synchronized method, or q is at a control point waiting for return.*

Note that for notification, we also require invariance of the assertions of threads waiting for notification. We do so, as notification is described by an auxiliary assignment executed by the notifier. That means, both the waiting and the notified status of a suspended thread are represented by a single control point in the wait method. The two statuses can be distinguished by the values of the wait and notified variables. The invariance of the precondition of the return statement in the wait method under the assignment in the notify method represents the notification process, whereas invariance of that assertion over assignments changing the lock represents the synchronization mechanism. Information about the lock value will be imported from the cooperation test as this information depends on the global behavior.

Example 4.3.2 *This example shows how the fact that at most one thread can own the lock of an object can be used to show mutual exclusion. We use the assertion* $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$ *for* $\mathsf{thread} \neq \mathsf{null} \wedge \mathsf{thread}(\mathsf{lock}) = \mathsf{thread}$*, where* $thread(lock)$ *is the first component of the lock value. Let* $\mathsf{free\_for}(\mathsf{thread}, \mathsf{lock})$ *be* $\mathsf{thread} \neq \mathsf{null} \wedge (\mathsf{owns}(\mathsf{thread}, \mathsf{lock}) \vee \mathsf{lock} = \mathsf{free})$.

Let q, given by $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$*, be an assertion at a control point and let* $\{p\}^{?call}\langle stm\rangle^{?call}$ *with* $p \stackrel{def}{=} \mathsf{free\_for}(\mathsf{thread}, \mathsf{lock})$ *be the callee observation at the beginning of a synchronized method in the same class. Note that the observation stm changes the lock value. The interference freedom condition* $\models_{\mathcal{L}} \{p \wedge q' \wedge \mathsf{interferes}(q, stm)\} stm \{q'\}$ *assures invariance of q under the observation stm. The assertions p and q' imply* $\mathsf{thread} = \mathsf{thread}'$*. The points at p and q can be simultaneously reached by the same thread only if q describes a point waiting for return. This fact is mirrored by the definition of the* interferes *predicate: If q is*

[5] This condition is not necessary for a minimal proof system, but reduces the number of verification conditions.

not at a control point waiting for return, then the antecedent of the condition evaluates to false. Otherwise, after the execution of the built-in augmentation $\mathsf{lock} := \mathsf{inc}(\mathsf{lock})$ *in stm we have* $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$*, i.e.,* $\mathsf{owns}(\mathsf{thread}', \mathsf{lock})$*, which was to be shown.*

The cooperation test

We extend the cooperation test for $Java_{conc}$ with the synchronization mechanism and with the invocation of the monitor methods. In the previous languages, the assertion comm expressed that the given statements indeed represent communicating partners. In the current language with monitor synchronization, communication is not always enabled. Thus the assertion comm has additionally to capture enabledness of the communication: In case of a synchronized method invocation, the lock of the callee object has to be free or owned by the caller. This is expressed by $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$, where thread is the caller thread, z' is the callee object, and where $\mathsf{thread}(z'.\mathsf{lock})$ is the first component of the lock value, i.e., the thread owning the lock of z'. For the invocation of the monitor methods we require that the executing thread is holding the lock. Returning from the wait method assumes that the thread has been notified and that the callee's lock is free.

Remember that the global invariant may only refer to instance variables whose values are modified by observations of communication or object creation only. Since the object-internal monitor signaling mechanism is represented by stand-alone auxiliary assignment, notification cannot affect the global invariant.

Definition 4.3.3 (Cooperation test: Communication) *A proof outline satisfies the* cooperation test for communication*, if the conditions of Definition 3.3.5 hold for the statements listed there with the exception of the* Call*-case, and additionally in the following cases:*

1. Call*: Invocations of non-synchronized methods m with* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *are treated as before. For all statements* $\{p_1\}\, u_{ret} := e_0.m(\vec{e})\ \{p_2\}^{!call}\langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_3\}^{wait}$ *(or such without receiving a value) in class c with* e_0 *of type* c'*, where method* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *of* c' *is synchronized with body* $\{q_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\, stm; \mathsf{return}\ e_{ret}$*, formal parameters* $\vec{u}$*, and local variables* $\vec{v}$ *except the formal parameters, Conditions 2.6 and 2.7 must hold with the following definitions: The callee class invariant is* $q_1 = I_{c'}$*. The assertion* comm *is given by* $E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$*. Furthermore,* f_{comm} *is* $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$*,* f_{obs1} *is given by* $z.\vec{y}_1 := \vec{E}_1(z)$*, and* f_{obs2} *is* $z'.\vec{y}_2' := \vec{E}_2'(z')$*.*
2. Call$_{monitor}$*: For* $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$*,* comm *is given by* $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$*.*
3. Return$_{wait}$*: For* $\{q_1\}\, \mathsf{return}_{getlock}\ \{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$ *in a* wait *method,* comm *is* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge z'.\mathsf{lock} = \mathsf{free} \wedge \mathsf{thread}' \in z'.\mathsf{notified}$*.*

Example 4.3.4 *Assume the invocation of a synchronized method m of a class c, where m of c has the body* $\langle stm \rangle^{?call}\{\mathsf{thread}(\mathsf{lock}) = \mathsf{thread}\}\ stm'; \mathsf{return}$. *Note that the built-in augmentation in stm sets the lock owner by the assignment* $\mathsf{lock} := \mathsf{inc}(\mathsf{lock})$. *Omitting irrelevant details again, the cooperation test requires* $\models_{\mathcal{G}} \{\mathsf{true}\} z'.\mathsf{lock} := \mathsf{inc}(z'.\mathsf{lock}) \{\mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}'\}$, *which holds by the definition of inc.*

Examples

Example 4.3.5 *The following proof outline is a producer-consumer implementation using synchronized methods and notification to assure mutual exclusion:*

```
class ProdCons{
    Int buffer;
    Bool written;

    Void sync produce(Int u){
        while (written) do wait() od;
        {¬written}
        buffer := u;
        written := true;
        notifyAll()
    }

    Int sync consume(){
        Int u;
        while (¬written) do wait() od;
        {written}
        u := buffer;
        written := false;
        notifyAll();
        return u
    }
}
```

The annotation expresses that prior to write access of the producer the shared buffer is not written (or already read); similarly for the consumer, prior to read access the buffer is written (and not yet read).

To prove invariance of the annotation we only have to show two local correctness conditions, stating that the loop-condition is false directly after exiting a while-loop. The interference freedom test does not generate any conditions, since the assertions in the synchronized methods are not at control points waiting for return. Finally, the pre- and postconditions of method bodies and method invocation statements are by definition true, and the class does not contain any object creation statement, such that also the cooperation test does not specify any conditions.

The conditions have been proven automatically in the theorem prover PVS.

Example 4.3.6 *Assume the annotated class below, which implements a simple account, offering interfaces for deposit and withdraw (see also Section 9.2.3). To assure that the balance x remains non-negative, the withdraw method is synchronized; implicitly, the balance does not get decreased between the evaluation of $x \geq i$ in the* withdraw *method and the withdrawal. The annotation expresses* *that for each class instance, under the assumption, that the methods* deposit *and* withdraw *are called with positive parameters only, the balance x has always a non-negative value, as stated in the class invariant. In the annotation we use the functions* $owns(thread, lock) \stackrel{def}{=} thread \neq null \wedge proj(lock, 1) = thread$ *and* $free\_for(thread, lock) \stackrel{def}{=} thread \neq null \wedge (proj(lock, 1) = thread \vee proj(lock, 1) = null)$ *for thread of type* Thread *and lock of type* $\mathsf{Thread} \times \mathsf{Int}$ *and with* $proj((v_1, \ldots, v_n), i) = v_i$ *for n-tuples* $(v_1, \ldots, v_n)$ *and* $1 \leq i \leq n$.

```
class Account{
    Int x;

    {x ≥ 0} //class invariant

    Void wait(){
        {false}?call {false}
        return {false}!ret
    }

    Void change_balance(int i){
        {i > 0 ∨ (x + i ≥ 0 ∧ owns(thread, lock))}
        x := x+i
        {i > 0 ∨ owns(thread, lock)}
    }

    Void deposit(int i){
        {i > 0}
        change_balance(i)
    }

    sync Void withdraw(int i){
        {free_for(thread, lock)}?call
        {i > 0 ∧ owns(thread, lock)}
        if (x ≥ i) {
            {x ≥ i ∧ i > 0 ∧ owns(thread, lock)}
            change_balance(-i);
            {i > 0}wait
            {owns(thread, lock)}
        } {owns(thread, lock)}
        return
        {owns(thread, lock)}!ret
    }
}
```

For the above proof outline 26 *verification conditions are generated (*4 *local correctness conditions,* 19 *interference freedom conditions, and* 3 *cooperation test conditions); see Section 9.2 for a detailed description of the conditions.*

4.4 Conclusions and related work

In this chapter we extended the concurrent language of the previous chapter by adding synchronization and reentrant monitors. Soundness and relative completeness are discussed in Chapter 6; the full proofs can be found in the appendix.

This work defines the first sound and relatively complete tool-supported assertional proof method for a multithreaded sublanguage of *Java* including its monitor discipline. In the following we discuss related work on the semantics of and on proof systems for multithreaded *Java* sublanguages with monitor synchronization.

4.4.1 Semantics

A denotational semantics is offered by Cenciarelli [Cen99] handling multithreading and exceptions. Cenciarelli et al. present in [CKRW97] a structural operational semantics of a concurrent *Java* sublanguage. This language includes dynamic creation of objects, blocks, and synchronization of threads. The authors start with an operational description for a sequential sublanguage. At the next stage, shared-memory interaction is described in terms of event spaces.

Based on [CKRW97], Cenciarelli et al. analyze the Java Memory Model (JMM) in [RKCW97]. They compare implementations of the memory model with and without prescient store actions (cf. Section 8.1). The authors prove that the two semantics coincide for properly synchronized programs. The structural operational semantics presented in [CKRW99] includes starting and stopping of threads, thread interaction via shared memory, monitoring and notification, and sequential control mechanisms such as exception handling and return statements. The operational semantics is parametric in the notion of event space. This allows different computational models to be obtained by modifying the well-formedness conditions on event spaces while leaving the operational rules untouched.

Coscia and Reggio [CR98, CR99] present an operational semantics of a multithreaded *Java* sublanguage. They discuss the memory model and state that correct use of synchronization guarantees that all processes agree on the values of shared variables.

Kassab and Greenwald [KG98] create a state-based abstraction of *Java* threads and security policies to study the enhanced *Java* 2 security model.

Attali et al. [ACR98] discuss a formal executable semantics of a concurrent subset of *Java* including inheritance, using the Centaur system. A formal executable specification of the concurrent Java Memory Model (JMM) is presented by Roychoudhury and Mitra [RM02]. Their specification is operational and uses guarded commands. They use their executable model also for verification.

Gontmakher et al. investigate the Java Memory Model in [GS00, GPS02]. They provide a trace-based characterization of the memory model, and compare it with other existing memory models. Pugh discusses the JMM in [Pug00]. Manson and Pugh [MP01a, MP01b] suggest alternative memory models with formal semantics, overcoming some of the problems raised by the JMM. Yang et al. [YGL02] give an implementation in the Uniform Memory Model. Another alternative is worked out by Maessen et al. [MAS00], using an enriched version of the Commit/Reconcile/Fence (CRF) memory model.

4.4.2 Proof system

Research on proof systems for concurrent class-based object-oriented languages with a monitor mechanism is very rare.

Buhr et al. [BFC95] give a survey about *monitors* in general, including proof-rules for various monitor semantics.

Verification is not restricted to *Java* source code: Moore et al. [Moo99, MKLP01, Moo02, LM03, MP03] show how the ACL2 theorem prover is capable not only of executing simple *Java* bytecode programs, but also of proving the correctness of such programs with respect to a specification. Their language covers inheritance, multithreading, and synchronization.

Chapter 5

Weakest precondition calculus

To increase readability, the verification conditions of the previous chapters have been formulated as standard Hoare triples. Our goal is to use a theorem prover to prove these conditions. Instead of implementing the semantics of Hoare triples within the theorem prover, we reformulate them into logical implications using a weakest precondition calculus. In this way we only have to implement the semantics of assertions within the theorem prover.

We first introduce substitutions in Section 5.1, before reformulating the verification conditions for $Java_{synch}$ in Section 5.2 into logical implications, using the substitutions. The proofs of the lemmas in this chapter can be found in Appendix A.1.

5.1 Substitution operations

The verification conditions defined in the next section involve three substitution operations: the local, the global, and the lifting substitutions. The lifting substitution is already defined in Section 2.3. The local substitution will be used to express the effect of assignments in local assertions. The global substitution is used similarly for global assertions.

The *local substitution* $p[\vec{e}/\vec{y}]$ is the standard capture-avoiding substitution, replacing in the local assertion p all free occurrences of the given distinct variables $\vec{y}$ by the local expressions $\vec{e}$. We apply the substitution also to local expressions. The following lemma expresses the standard property of the above substitution, relating it to state update. The relation between substitution and update formulated in the lemma asserts that $p[\vec{e}/\vec{y}]$ is the *weakest precondition* of p with respect to the assignment $\vec{y} := \vec{e}$. The lemma is formulated for assertions, but the same property holds for expressions.

Lemma 5.1.1 (Local substitution) *For arbitrary logical environments ω and instance local states (σ_{inst}, τ) we have*

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p[\vec{e}/\vec{y}] \qquad \textit{iff} \qquad \omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}] \models_{\mathcal{L}} p\,.$$

The effect of assignments is expressed on the global level by the *global substitution* $P[\vec{E}/z.\vec{x}]$, which replaces in the global assertion P the instance variables $\vec{x}$ of the object referred to by z by the global expressions $\vec{E}$. To accommodate properly the effect of assignments, though, we must not only syntactically replace the occurrences $z.x_i$ of the instance variables, but also all their *aliases* $E'.x_i$, when z and the result of the substitution applied to E' refer to the same object. As the aliasing condition cannot be checked syntactically, we define the main case of the substitution by a conditional expression [AdB93]:

$$(E'.x_i)[\vec{E}/z.\vec{x}] = \ (\mathsf{if}\ \ E'[\vec{E}/z.\vec{x}] = z\ \ \mathsf{then}\ \ E_i\ \ \mathsf{else}\ (E'[\vec{E}/z.\vec{x}]).x_i\ \ \mathsf{fi})\ .$$

This substitution is extended to global assertions homomorphically. We will also use the substitution $P[\vec{E}/z.\vec{y}]$ for arbitrary variable sequences $\vec{y}$ possibly containing logical variables, whose semantics is defined by the simultaneous substitutions $[\vec{E}_x/z.\vec{x}]$ and $[\vec{E}_u/\vec{u}]$, where $\vec{x}$ and $\vec{u}$ are the sequences of the instance and logical variables[1] of $\vec{y}$, and $\vec{E}_x$ and $\vec{E}_u$ the corresponding subsequences of $\vec{E}$ and $[\vec{E}_u/\vec{u}]$ is the usual capture-avoiding substitution like in the local substitution; if only logical variables are substituted, we simply write $P[\vec{E}/\vec{u}]$. That the substitution accurately catches the semantical update, and thus represents the weakest precondition relation, is expressed by the following lemma:

Lemma 5.1.2 (Global substitution) *For arbitrary global states σ and logical environments ω referring only to values existing in σ we have*

$$\omega, \sigma \models_{\mathcal{G}} P[\vec{E}/z.\vec{y}] \qquad \textit{iff} \qquad \omega', \sigma' \models_{\mathcal{G}} P\ ,$$

where $\omega' = \omega[\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]$ and $\sigma' = \sigma[[\![z]\!]_{\mathcal{G}}^{\omega,\sigma}.\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]$.

5.2 Verification conditions

In the local verification conditions, the effect of an assignment $\vec{y} := \vec{e}$ is expressed by substituting $\vec{e}$ for $\vec{y}$ in the assertions. In the global conditions of the cooperation test, the effect of communication, changing local states only, is expressed by simultaneously substituting those variables which will store the result by the communicated values. I.e., for the case of method call, the formal parameters are replaced by the actual ones expressed in the global language. The effect of the caller observation $\langle \vec{y} := \vec{e}\rangle^{!call}$ upon a global assertion P is expressed by the substitution $P[\vec{E}(z)/z.\vec{y}]$, where z represents the caller. The effect of the callee-observation is handled similarly. Note the order: first communication takes place, followed by the sender, and then the receiver observation.

[1] Local variables are viewed as logical ones in the global assertion language.

To describe the joint effect, we first have to substitute for the receiver, then for the sender observation, and, finally, for communication. For a method call, we additionally have to substitute for the initialization of the local variables.

For readability, in the following definitions we use the notation $p \circ f$ with $f = [\vec{e}/\vec{y}]$ for the substitution $p[\vec{e}/\vec{y}]$; we use a similar notation for global assertions. Note that the substitution binds stronger than logical operators.

Definition 5.2.1 (Initial correctness) *A proof outline is* initially correct, *if*

$$\begin{aligned} \models_{\mathcal{G}} \quad & \mathsf{InitState}(z) \wedge (\forall z'.\ z' = \mathsf{null} \vee z = z') \rightarrow \\ & P_2(z) \circ f_{init} \wedge (\,GI \wedge P_3(z) \wedge I_c(z)) \circ f_{obs} \circ f_{init}\ , \end{aligned} \tag{5.1}$$

where c is the main class, $\{p_2\}^{?call}\langle\vec{y}_2 := \vec{e}_2\rangle^{?call}\{p_3\}\ stm;$ return *is the body and $\vec{v}$ the local variables of the* run *method of c, $z \in LVar^c$, and $z' \in LVar^{\mathsf{Object}}$. The global assertion* InitState *is defined on page 32. Furthermore,*

$$\begin{aligned} f_{init} &= [z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]\ ,\ \textit{and} \\ f_{obs} &= [\vec{E}_2(z)/z.\vec{y}_2]\,. \end{aligned}$$

Definition 5.2.2 (Local correctness: Assignment) *A proof outline is* locally correct*, if for all multiple assignments $\{p_1\}\ \vec{y} := \vec{e}\ \{p_2\}$ in class c, which is not the observation of communication or object creation,*

$$\models_{\mathcal{L}} \quad p_1 \wedge I_c \rightarrow p_2 \circ f_{ass}\ , \tag{5.2}$$

with $f_{ass} = [\vec{e}/\vec{y}]$.

Definition 5.2.3 (Interference freedom) *A proof outline is* interference free*, if for all classes c, and for all multiple assignments $\vec{y} := \vec{e}$ with precondition p in c,*

$$\models_{\mathcal{L}} \quad p \wedge I_c \rightarrow I_c \circ f_{ass}\ , \tag{5.3}$$

with $f_{ass} = [\vec{e}/\vec{y}]$. Furthermore, for all assertions q at control points in c, such that either not both p and q occur in a synchronized method, or q is at a control point waiting for return,

$$\models_{\mathcal{L}} \quad p \wedge q' \wedge I_c \wedge \mathsf{interferes}(q, \vec{y} := \vec{e}) \rightarrow q' \circ f_{ass}\ . \tag{5.4}$$

with the assertion interferes *as defined on page 57.*

Definition 5.2.4 (Cooperation test: Communication) *A proof outline satisfies the* cooperation test for communication*, if*

$$\begin{aligned} \models_{\mathcal{G}} \quad & GI \wedge P_1(z) \wedge I_c(z) \wedge Q_1'(z') \wedge I_{c'}(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null} \rightarrow \\ & (P_2(z) \wedge Q_2'(z')) \circ f_{comm} \wedge \\ & (\,GI \wedge P_3(z) \wedge Q_3'(z')) \circ f_{obs2} \circ f_{obs1} \circ f_{comm} \end{aligned} \tag{5.5}$$

holds for distinct fresh logical variables $z \in LVar^c$ and $z' \in LVar^{c'}$, in the following cases:

1. (a) CALL: *For all calls* $\{p_1\}\, u_{ret} := e_0.m(\vec{e})\ \{p_2\}^{!call}\langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_3\}^{wait}$ *(or such without receiving a value) in class c with* e_0 *of type* c'*, where method* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *of* c' *is synchronized with body* $\{q_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\, stm; \mathsf{return}\ e_{ret}$*, formal parameters* $\vec{u}$*, and local variables* $\vec{v}$ *except the formal parameters. The callee class invariant is* $q_1 = I_{c'}$*. The assertion* comm *is given by* $E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$*. Furthermore,* $f_{comm} = [\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$*,* $f_{obs1} = [\vec{E}_1(z)/z.\vec{y}_1]$*,* $f_{obs2} = [\vec{E}'_2(z')/z'.\vec{y}'_2]$*. If* m *is not synchronized,* $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *in* comm *is dropped.*

   (b) CALL$_{monitor}$*: For* $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$*,* comm *is given by* $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$*.*

   (c) CALL$_{start}$*: For* $m = \mathsf{start}$*,* comm *is* $E_0(z) = z' \wedge \neg z'.\mathsf{started}$*, where* $\{q_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\, stm; \mathsf{return}$ *is the body of the* run *method of* c'*.*

   (d) CALL$^{skip}_{start}$*: For* $m = \mathsf{start}$*, additionally, (5.5) must hold with* comm *given by* $E_0(z) = z' \wedge z'.\mathsf{started}$*,* $q_2 = q_3 = \mathsf{true}$*, and* f_{comm} *and* f_{obs2} *are the identity functions.*

2. (a) RETURN*: For all method call statements* $u_{ret} := e_0.m(\vec{e})\ \langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_1\}^{wait}\{p_2\}^{?ret}\langle \vec{y}_4 := \vec{e}_4\rangle^{?ret}\{p_3\}$ *(or such without receiving a value) occurring in c with* e_0 *of type* c'*, such that method* $m(\vec{u})$ *of* c' *has the return statement* $\{q_1\}\ \mathsf{return}\ e_{ret}\ \{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$*, Equation (5.5) must hold with* comm *given by* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$*, and where* $f_{comm} = [E'_{ret}(z')/u_{ret}]$*,* $f_{obs1} = [\vec{E}'_3(z')/z'.\vec{y}'_3]$*, and* $f_{obs2} = [\vec{E}_4(z)/z.\vec{y}_4]$*.*

   (b) RETURN$_{wait}$*: For* $\{q_1\}\ \mathsf{return}_{getlock}\ \{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$ *in a* wait *method,* comm *is* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge z'.\mathsf{lock} = \mathsf{free} \wedge \mathsf{thread}' \in z'.\mathsf{notified}$*.*

   (c) RETURN$_{run}$*: For* $\{q_1\}\ \mathsf{return}\ \{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$ *occurring in a* run *method,* $p_1 = p_2 = p_3 = \mathsf{true}$*,* $\mathsf{comm} = \mathsf{true}$*, and furthermore* f_{comm} *and* f_{obs2} *the identity function.*

Definition 5.2.5 (Cooperation test: Instantiation) *A proof outline satisfies the* cooperation test for object creation*, if for all classes* c' *and statements* $\{p_1\}\, u := \mathsf{new}^c\ \{p_2\}^{new}\langle \vec{y} := \vec{e}\rangle^{new}\{p_3\}$ *in* c'*:*

$$\models_{\mathcal{G}} \quad z \neq \mathsf{null} \wedge z \neq u \wedge \exists z'.\ \big(\mathsf{Fresh}(z', u) \wedge (GI \wedge (\exists u.\ P_1(z)) \wedge I_{c'}(z))\ \downarrow z'\big) \rightarrow P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z)) \circ f_{obs}\ , \tag{5.6}$$

with $z \in LVar^{c'}$ *and* $z' \in LVar^{\mathsf{list\ Object}}$ *fresh,* $f_{obs} = [\vec{E}(z)/z.\vec{y}]$*, and* Fresh *and* $\downarrow$ *as defined in Section 2.4.2 on page 43.*

5.3 Conclusions

This chapter reformulates the Hoare-style verification conditions for the parallel language with monitor synchronization to logical implications. The effect of assignments is described by substitutions.

The *Verger* tool generates not only the conditions, but applies also the substitutions to the assertions. Thus the verification conditions are logical implications. Consequently, we only need to encode the semantics of the assertion language in the theorem prover. This simple representation of the verification conditions in the theorem prover increases the automation of the proofs.

The more semantically-oriented approaches based on the global store model [AL97, JKW03, vON02, PHM99] require an explicit encoding of the semantics of assignments.

Chapter 6

Soundness and completeness

This section discusses soundness and relative completeness proofs for the proof method of Section 5.2; these proofs are listed in the appendix.

Given a program together with its annotation, the proof system stipulates a number of verification conditions for the various types of assertions and program constructs. *Soundness* of the proof system means that for a proof outline satisfying the verification conditions, all configurations reachable in the operational semantics satisfy the given assertions. *Completeness*, conversely, means that if a program does satisfy an annotation, this fact is provable.

Gödel's *Incompleteness Theorem* (1931) states that there is no proof system where all valid assertions are provable. This implies that Hoare logic is not complete either. However, we can show *relative* completeness of Hoare logic, which means completeness relative to the underlying logic: If we assume that we have proofs for the assertions, then Hoare logic is complete. In the following, if we talk about completeness, we always mean relative completeness.

Cook introduced the notion of relative completeness in [Coo78], and proves that the Hoare logic of while programs [Hoa69] is sound and relatively complete. In [TZ88] Tucker and Zucker extend Cook's result to iteration and recursion. Apt [Apt81a, Apt83] has shown that both for shared variable concurrency and for synchronous message passing, the completeness proofs have to be based on merging lemmas, which he introduced.

The survey of various results concerning Hoare's approach to proving program correctness is presented in [Apt81b].Emphasis is placed on the soundness and completeness issues. [dRdBH$^+$01]is a systematic and comprehensive introduction both to compositional and to noncompositional proof methods for the state-based verification of concurrent programs, including soundness and completeness results.

For convenience, let us introduce the following notations: Given a program *prog*, we will write φ_{prog} or just φ for its annotation, and write *prog* $\models \varphi$, if *prog* satisfies all requirements stated in the assertions, and $prog' \vdash \varphi'$, if $prog'$ with annotation φ' satisfies the verification conditions of the proof system:

Definition 6.0.1 *Given a program prog with annotation φ, then $prog \models \varphi$ iff for all reachable configurations $\langle T, \sigma \rangle$ of prog, for all $(\alpha, \tau, stm) \in T$, and for all logical environments ω referring only to values existing in σ:*

1. $\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} pre(stm)$*, and*

2. $\omega, \sigma \models_{\mathcal{G}} GI$.

Furthermore, for all classes c, objects $\beta \in Val^c(\sigma)$, and local states τ':

3. $\omega, \sigma(\beta), \tau' \models_{\mathcal{L}} I_c$.

For proof outlines, we write $prog' \vdash \varphi'$ iff $prog'$ with annotation φ' satisfies the verification conditions of the proof system.

In the following sections we discuss the basic ideas of the soundness and relative completeness proofs. The formal proofs can be found in the appendix.

6.1 Soundness

Soundness, as mentioned, means that all reachable configurations do satisfy their assertions for an annotated program that has been verified using the proof conditions. Soundness of the method is proved by a straightforward, albeit tedious, induction on the number of computation steps.

Before embarking upon the soundness formulation and its proof, we need to clarify the connection between the original program and the proof outline, i.e., the one extended by auxiliary variables, and decorated with assertions. The transformation is done for the sake of verification, only, and as far as the unaugmented portion of the states and the configurations is concerned, the behavior of the original and the transformed program are the same.

To make the connection between original program and the proof outline precise, we define a projection operation $\downarrow prog$, that erases all additions of the transformation. So let $prog'$ be a proof outline for *prog*, and $\langle T', \sigma' \rangle$ a global configuration of $prog'$. Then $\sigma' \downarrow prog$ is defined by removing all auxiliary instance variables from the instance state domains. For the set of thread configurations, $T' \downarrow prog$ is given by restricting the domains of the local states to non-auxiliary variables and removing all augmentations. Additionally, for local configurations $(\alpha, \tau, \mathsf{return}_{getlock}\langle stm \rangle^{!ret}) \in T'$, if the executing thread is in the wait set, i.e., if $(\tau(\mathsf{thread}), n) \in \sigma'(\alpha)(\mathsf{wait})$ for some n, then the statement $\mathsf{return}_{getlock}$ gets replaced by $?\mathsf{signal}; \mathsf{return}_{getlock}$. Furthermore, for local configurations $(\alpha, \tau, stm; \mathsf{return}\ \langle stm' \rangle^{!ret}) \in T'$ with $stm \neq \epsilon$ an auxiliary assignment in the notify or the notifyAll method, the auxiliary assignment *stm* gets replaced by !signal and !signal_all, respectively. The following lemma expresses that this transformation does not change the behavior of programs:

Lemma 6.1.1 *Let $prog'$ be a proof outline for a program $prog$. Then $\langle T, \sigma\rangle$ is a reachable configuration of $prog$ iff there exists a reachable configuration $\langle T', \sigma'\rangle$ of $prog'$ with $\langle T' \downarrow prog, \sigma' \downarrow prog\rangle = \langle T, \sigma\rangle$.*

The augmentation introduced a number of specific built-in auxiliary variables that reflect the predicates used in the semantics. That the semantics is faithfully represented by the variables is formulated in the following lemmas. For the variables thread and conf we show that their values are unique identifiers:

Lemma 6.1.2 (Identification) *Let $\langle T, \sigma\rangle$ be a reachable configuration of a proof outline. Then:*

1. *for all stacks ξ and ξ' in T and for all local configurations $(\alpha, \tau, stm) \in \xi$ and $(\alpha', \tau', stm') \in \xi'$ we have $\tau(\mathsf{thread}) = \tau'(\mathsf{thread})$ iff $\xi = \xi'$, and*
2. *for each stack $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n)$ in T and indices $0 \le i, j \le n$,*
   (a) $\tau_i(\mathsf{thread}) = \alpha_0$;
   (b) $i < j$ *and* $\alpha_i = \alpha_j$ *implies* $\tau_i(\mathsf{conf}) < \tau_j(\mathsf{conf}) < \sigma(\alpha_i)(\mathsf{counter})$,
   (c) $0 < j$ *implies* $\tau_j(\mathsf{caller}) = (\alpha_{j-1}, \tau_{j-1}(\mathsf{conf}), \tau_{j-1}(\mathsf{thread}))$, *and*
   (d) $\mathsf{proj}(\tau_0(\mathsf{caller}), 3) \neq \tau_0(\mathsf{thread})$,

where $proj(v, i)$ is the ith component of the tuple v.

The following lemma states that the lock ownership, and the *wait* and *notified* sets of the semantics are correctly represented by the variables lock, wait, and $\mathsf{notified}$. Furthermore, the lemma assures disjunctness of the sequences stored in the wait and $\mathsf{notified}$ variables; if the order of the elements is unimportant, we use set notation for their values.

Lemma 6.1.3 (Lock, Wait, Notify) *Let $\langle T, \sigma\rangle$ be a reachable configuration of a proof outline for the original program $prog$, $\alpha \in Val(\sigma)$ an object identity, and let $\xi = (\alpha_0, \tau_0, stm_0) \circ \xi' \in T$. Let furthermore n be the number synchronized method executions of ξ in α, i.e., $n = |\{(\alpha, \tau, stm) \in \xi \mid stm \ synchr.\}|$. Then:*

1. (a) $\neg owns(T \downarrow prog, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{lock}) = free$
   (b) $owns(\xi \downarrow prog, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{lock}) = (\alpha_0, n)$
2. (a) $\xi \in wait(T \downarrow prog, \alpha)$ *iff* $(\alpha_0, n) \in \sigma(\alpha)(\mathsf{wait})$
   (b) $\xi \in notified(T \downarrow prog, \alpha)$ *iff* $(\alpha_0, n) \in \sigma(\alpha)(\mathsf{notified})$
   (c) $proj(\sigma(\alpha)(\mathsf{wait})[i], 1) = proj(\sigma(\alpha)(\mathsf{wait})[j], 1)$ *implies* $i = j$
   (d) $proj(\sigma(\alpha)(\mathsf{notified})[i], 1) = proj(\sigma(\alpha)(\mathsf{notified})[j], 1)$ *implies* $i = j$
   (e) *if* $(\alpha_0, m) \in \sigma(\alpha)(\mathsf{wait})$ *or* $(\alpha_0, m) \in \sigma(\alpha)(\mathsf{notified})$ *then* $m = n$
   (f) $\sigma(\alpha)(\mathsf{wait}) \cap \sigma(\alpha)(\mathsf{notified}) = \emptyset$,

where $s[i]$ *is the* i*th element of the sequence* s*.*

Finally, the auxiliary instance variable $\mathsf{started}$ of an object correctly stores if the thread of the object is already started or not:

Lemma 6.1.4 (Started) *For all reachable configurations* $\langle T, \sigma\rangle$ *of a proof outline for a program prog, and all objects* $\alpha \in Val(\sigma)$*, we have* $started(T \downarrow prog, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{started})$*.*

Let *prog* be a program with annotation φ, and $prog'$ a corresponding proof outline with annotation φ'. Let GI' be the global invariant of φ', I'_c denote its class invariants, and for an assertion p of φ let p' denote the assertion of φ' associated with the same control point. We write $\models \varphi' \to \varphi$ iff $\models_{\mathcal{G}} GI' \to GI$, $\models_{\mathcal{L}} I'_c \to I_c$ for all classes c, and $\models_{\mathcal{L}} p' \to p$, for all assertions p of φ associated with some control point. To give meaning to the auxiliary variables, the above implications are evaluated in the context of states of the augmented program. The following theorem states the soundness of the proof method.

Theorem 6.1.5 (Soundness) *Let* $prog'$ *be a proof outline with annotation* $\varphi_{prog'}$*.*

$$\textit{If} \quad prog' \vdash \varphi_{prog'} \quad \textit{then} \quad prog' \models \varphi_{prog'} \ .$$

The soundness proof consists basically of an induction argument on the length of computation, simultaneously on all three parts from Definition 6.0.1. For the inductive step, we assume that the verification conditions are satisfied and assume a reachable configuration satisfying the annotation. We make case distinction on the syntax of the next computation step: If the computation step executes an assignment, then we use the local correctness conditions to prove inductivity of the executing local configuration's properties, and the interference freedom test for all other local configurations and the class invariants. For communication, invariance for the executing partners and the global invariant is shown using the cooperation test for communication. Communication itself does not affect the global state; invariance of the remaining properties under the corresponding observations is shown again with the help of the interference freedom test. Finally, for object creation, invariance for the global invariant, the creator local configuration, the created object's class invariant is assured by the conditions of the cooperation test for object creation; all other properties are shown to be invariant using the interference freedom test.

Theorem 6.1.5 is formulated for reachability of augmented programs. With the help of Lemma 6.1.1, we immediately get:

Corollary 6.1.6 *If* $prog' \vdash \varphi_{prog'}$ *and* $\models \varphi_{prog'} \to \varphi_{prog}$*, then* $prog \models \varphi_{prog}$*.*

6.2 Completeness

Next we, conversely, show that if a program satisfies the requirements asserted in its proof outline, then this is indeed provable, i.e., then there exists a proof outline which can be shown to hold and which implies the given one:

$$\forall prog.\ prog \models \varphi_{prog} \ \Rightarrow\ \exists prog'.\ prog' \vdash \varphi_{prog'} \wedge\ \models \varphi_{prog'} \rightarrow \varphi_{prog}\ .$$

Given a program satisfying an annotation $prog \models \varphi_{prog}$, the consequent can be uniformly shown, i.e., independently of the given assertional part φ_{prog}, by instantiating $\varphi_{prog'}$ to the strongest annotation still provable, thereby discharging the last clause $\models \varphi_{prog'} \rightarrow \varphi_{prog}$. Since the strongest annotation still satisfied by the program corresponds to reachability, the key to completeness is to:

1. augment each program with enough information (see Definition 6.2.1 below), to be able to
2. express reachability in the annotation, i.e., annotate the program such that a configuration satisfies its local and global assertions exactly if reachable (see Definition 6.2.2 below), and, finally,
3. to show that this augmentation indeed satisfies the verification conditions.

We begin with the augmentation, using the transformation from Section 4.3 as starting point, where the programs are augmented with the specific auxiliary variables.

In the following we define an augmentation which allows to formulate the reachability annotation. In this thesis we do not focus on a minimal augmentation, that means, we record also information in additional auxiliary variables which would not be necessary for the reachability annotation but which simplify the formalization and the proofs. For example, in the paragraph below we introduce a location counter, which is not necessary but which simplifies the proofs by the recorded information about the control point of the executing thread. Also in the history variables introduced further below we record more information than it would be necessary for the formalization of the reachability annotation. A more abstract formulation builds a topic for future work.

To facilitate reasoning, we introduce an additional auxiliary local variable loc, which stores the current control point of the execution of a local configuration. Given a function which assigns to all control points unique location labels, we extend each assignment with the update $\mathsf{loc} := l$, where l is the label of the control point after the given occurrence of the assignment. Also unobserved statements are extended with the update. We write $l \equiv stm$ if l represents the control point in front of stm.

The standard method for obtaining a completeness augmentation is to add information into the states about the way how it has been reached, i.e., to add the *history* of the computation leading to the configuration. This information is recorded using history variables.

The assertion language is split into a local and a global level, and, likewise, the proof system is tailored to separate local proof obligations from global ones to obtain a modular proof system. The history will be recorded in instance variables, and thus each instance can keep track only of its own past. To mirror the split into a local and a global level in the proof system, the history per instance is recorded separately for *internal* and *external* behavior. The sequence of internal state changes local to that instance is recorded in the *local* history h_{inst} and the external behavior in the *communication* history h_{comm}.

The local history keeps track of the state updates. We store in the local history the updated local and instance states of the executing local configuration and the object in which the execution takes place. Note that the local history stores also the values of the built-in auxiliary variables, and thus the identities of the executing thread and the executing local configuration.

The communication history contains information about the kind of communication, the communicated values, and the identity of the communication partners involved. For communication, we distinguish as cases object creation, ingoing and outgoing method calls, and, likewise, ingoing and outgoing communication for the return value. We use the set $\bigcup_{c\in\mathcal{C}}\{\mathsf{new}^c\}\cup\bigcup_{m\in\mathcal{M}}\{!m,?m\}\cup\{!\mathsf{return},?\mathsf{return}\}$ of constants for this purpose, where $\mathcal{C}$ and $\mathcal{M}$ are the sets of all class and method names, respectively. Notification does not update the communication history, since it is object-internal computation. For the same reason, we do not record self-communication in h_{comm}. Note in passing that the information stored in the communication history matches exactly the information needed to decorate the transitions in order to obtain a compositional variant of the operational semantics of Section 4.2. See [ÁdBdRS04a] for such a compositional semantics.

Definition 6.2.1 (Augmentation with histories) *Every class is further extended by two auxiliary instance variables h_{inst} and h_{comm}, both initialized to the empty sequence. They are updated as follows:*

1. *Each multiple assignment $\vec{y} := \vec{e}$ in each class c that is not the observation of a method call or of the reception of a return value is extended with*

$$\mathsf{h}_{inst} := \mathsf{h}_{inst} \circ ((\vec{x},\vec{v})[\vec{e}/\vec{y}]) \ ,$$

   where $\vec{x}$ are the instance variables of class c containing also h_{comm} but without h_{inst}, and $\vec{v}$ are the local variables. Observations $\vec{y} := \vec{e}$ of $u_{ret} := e_0.m(\vec{e}')$ and of the corresponding reception of the return value are extended with the assignment

$$\mathsf{h}_{inst} := \mathsf{if}\ (e_0 = \mathsf{this})\ \mathsf{then}\ \mathsf{h}_{inst}\ \mathsf{else}\ \mathsf{h}_{inst} \circ ((\vec{x},\vec{v})[\vec{e}/\vec{y}])\ \mathsf{fi}\ ,$$

   instead, if $m \neq \mathsf{start}$. For $e_0.\mathsf{start}(\vec{e}'); \langle \vec{y} := \vec{e}\rangle^{!call}$ we use the same update with the condition $e_0 = \mathsf{this}$ replaced by $e_0 = \mathsf{this} \wedge \neg\mathsf{started}$.

2. *Every communication and object creation statement is observed by*

$$\begin{aligned}\mathsf{h}_{comm} \quad := \quad & \mathsf{if}\ (\mathsf{partner} = \mathsf{this})\ \mathsf{then}\ \mathsf{h}_{comm}\ \mathsf{else} \\ & \mathsf{h}_{comm} \circ (\mathsf{sender}, \mathsf{receiver}, \mathsf{values})\ \mathsf{fi}\ ,\end{aligned}$$

where the expressions partner*,* sender*,* receiver*, and* values *depend on the kind of communication as follows:*

communication	partner	sender	receiver	values
$u := \mathsf{new}^c$	null	this	null	$\mathsf{new}^c\, u, \mathsf{thread}$
$u_{ret} := e_0.m(\vec{e})$	e_0	this	e_0	$!m(\vec{e})$
receive return	e_0	e_0	this	$?\,\mathsf{return}\ u_{ret}, \mathsf{thread}$
receive call $m(\vec{u})$	caller_obj	caller_obj	this	$?m(\vec{u})$
return e_{ret}	caller_obj	this	caller_obj	$!\,\mathsf{return}\ e_{ret}, \mathsf{thread}$

with caller_obj *given by the first component of the variable* caller*.*

In the update of the history variable h_{inst}, the expression $(\vec{x}, \vec{u})[\vec{e}/\vec{y}]$ identifies the active thread and local configuration given by the local variables thread and conf, and specifies its instance local state after the execution of the assignment. Note that especially the values of the auxiliary variables introduced in the augmentation are recorded in the local history. In the following we will also write (σ_{inst}, τ), when referring to elements of h_{inst}.

Note furthermore that the communication history records also the identities of the communicating threads in values.

Next we introduce the annotation for the augmented program.

Definition 6.2.2 (Reachability annotation) *We define the following annotation for the augmented program:*

1. $\omega, \sigma \models_{\mathcal{G}} GI$ *iff there exists a reachable* $\langle T, \sigma' \rangle$ *such that* $Val(\sigma) = Val(\sigma')$*, and for all* $\alpha \in Val(\sigma)$*,* $\sigma(\alpha)(\mathsf{h}_{comm}) = \sigma'(\alpha)(\mathsf{h}_{comm})$*.*

2. *For each class* c*, let* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} I_c$ *iff there is a reachable* $\langle T, \sigma \rangle$ *such that* $\sigma(\alpha) = \sigma_{inst}$*, where* $\alpha = \sigma_{inst}(\mathsf{this})$*. For each class* c *and method* m *of* c*, the pre- and postconditions of* m *are given by* I_c*.*

3. *For assertions at control points,* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(stm)$ *iff there is a reachable* $\langle T, \sigma \rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$ *for* $\alpha = \sigma_{inst}(\mathsf{this})$*, and such that* $(\alpha, \tau, stm; stm') \in T$*.*

4. *For preconditions* p *of observations of communication or object creation, let* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ *iff there is a reachable* $\langle T, \sigma \rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$ *for* $\alpha = \sigma_{inst}(\mathsf{this})$*, and with* $(\alpha, \tau', stm; stm') \in T$ *enabled to communicate resulting in the local state* τ *directly after communication, where* stm *is the corresponding communication statement.*

   For observing the reception of a method call, instead of the existence of the enabled $(\alpha, \tau', stm; stm') \in T$*, we require that a call of the method of* α *is enabled in* $\langle T, \sigma \rangle$ *with resulting callee local state* τ *directly after communication*[1]*.*

---

[1]For the precondition of the observation stm at the beginning of the run method of the main class, $\langle T, \sigma \rangle$ can also be the initial configuration before the execution of the observation stm.

It can be shown that these assertions are expressible in the assertion language (see [TZ88]). Expressing reachability in the annotation relies heavily on quantification over sequences. The augmented program together with the above annotation build a proof outline that we express by *prog*$'$.

What remains to be shown for relative completeness is that the proof outline *prog*$'$ indeed satisfies the verification conditions of the proof system. Initial and local correctness are straightforward.

Completeness for the interference freedom test and the cooperation test are more complex, since, unlike initial and local correctness, the verification conditions in these cases mention more than one local configuration in their respective antecedents. Now, the reachability assertions of *prog*$'$ guarantee that, when satisfied by an instance local state, there *exists* a reachable global configuration responsible for the satisfaction. So a crucial step in the completeness proof for interference freedom and the cooperation test is to show that individual reachability of two local configurations in the same instance state implies that they are reachable in a *common* computation. This is also the key property for the history variables: They record enough information such that they allow one to uniquely determine the way a configuration has been reached; in the case of the instance history uniqueness applies only as far as the chosen instance is concerned. This property is stated formally in the following local merging lemma.

Lemma 6.2.3 (Local merging lemma) *Assume two reachable global configurations* $\langle T_1, \sigma_1 \rangle$ *and* $\langle T_2, \sigma_2 \rangle$ *of* *prog*$'$ *and* $(\alpha, \tau, stm) \in T_1$ *with* $\alpha \in Val(\sigma_1) \cap Val(\sigma_2)$. *Then* $\sigma_1(\alpha)(\mathsf{h}_{inst}) = \sigma_2(\alpha)(\mathsf{h}_{inst})$ *implies* $(\alpha, \tau, stm) \in T_2$.

For completeness of the cooperation test, connecting two possibly different instances, we need an analogous property for the communication histories. Arguing on the global level, the cooperation test can assume that two control points are individually reachable but agree on the communication histories of the objects. This information must be enough to ensure common reachability. Such a common computation can be constructed, since the internal computations of different objects are independent from each other, i.e., in a global computation, the local behavior of an object is interchangeable, as long as the external behavior does not change. This leads to the following lemma:

Lemma 6.2.4 (Global merging lemma) *Assume two reachable global configurations* $\langle T_1, \sigma_1 \rangle$ *and* $\langle T_2, \sigma_2 \rangle$ *of* *prog*$'$ *and* $\alpha \in Val(\sigma_1) \cap Val(\sigma_2)$ *with the property* $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm})$. *Then there exists a reachable configuration* $\langle T, \sigma \rangle$ *with* $Val(\sigma) = Val(\sigma_2)$, $\sigma(\alpha) = \sigma_1(\alpha)$, *and* $\sigma(\beta) = \sigma_2(\beta)$ *for all* $\beta \in Val(\sigma_2) \backslash \{\alpha\}$.

Note that together with the local merging lemma this implies that all local configurations in $\langle T_1, \sigma_1 \rangle$ executing in α and all local configurations in $\langle T_2, \sigma_2 \rangle$ executing in $\beta \neq \alpha$ are contained in the commonly reached configuration $\langle T, \sigma \rangle$.

This brings us to the completeness result:

Theorem 6.2.5 (Relative completeness) *For a program prog, the proof outline prog′ satisfies the verification conditions of the proof system from Section 5.2.*

Chapter 7

Proving deadlock freedom

The previous chapters described a proof system which can be used to prove safety properties of $Java_{synch}$ programs. In this section we show how to apply the proof system to prove *deadlock freedom*.

7.1 Expressing deadlock freedom

A system of processes is in a deadlocked configuration if no one of them is enabled to compute but not yet all processes are terminated. A typical deadlock situation can occur, if two threads t_1 and t_2 both try to reserve the locks of two objects o_1 and o_2, but in reverse order: t_1 first applies for access to the synchronized methods of o_1, and then for those of o_2, while t_2 first collects the lock of o_2, and tries to become the lock owner of o_1. Now, it can happen, that t_1 gets the lock of o_1, t_2 gets the lock of o_2, and both are waiting for the other lock, which will never become free. Another typical source of deadlock situations are threads which suspended themselves by calling wait and which will never get notified.

So, what kind of statements can be disabled and under which conditions? The important cases, to which we restrict, are:

- The invocation of synchronized methods, if the lock of the callee object is neither free nor owned by the executing thread.
- If a thread tries to invoke a monitor method of an object whose lock it does not own.
- If a thread tries to return from a wait method, but either the lock is not free or the thread is not yet notified.

To be precise, the semantics specifies method calls to be disabled also if the callee object is the empty reference. However, we do not deal with this case; it can be excluded in the preconditions by stating that the callee object is not *null*.

Assume a proof outline with global invariant GI. For a logical variable z of type Object, let $I(z) = I[z/\mathsf{this}]$ be the class invariant of z expressed on the global level. Let the assertion $\mathsf{terminated}(z)$ express that the thread of z is already terminated. Formally, we define $\mathsf{terminated}(z)$ by $\exists \vec{v}.\, q[z/\mathsf{thread}][z/\mathsf{this}]$, where q is the postcondition of the run method of z, and $\vec{v}$ its local variables. For assertions p in an object represented by z' let furthermore $\mathsf{blocked}(z, z', p)$ express that the thread of z is disabled in the object z' at the control point described by p. Formally, we define $\mathsf{blocked}(z, z', p)$ by:

- $\exists \vec{v}.\, p[z/\mathsf{thread}][z'/\mathsf{this}] \wedge e_0.\mathsf{lock} \neq \mathsf{free} \wedge \mathsf{thread}(e_0.\mathsf{lock}) \neq \mathsf{thread}$ if p is the precondition of a call invoking a synchronized method of e_0,

- $\exists \vec{v}.\, p[z/\mathsf{thread}][z'/\mathsf{this}] \wedge \mathsf{thread}(e_0.\mathsf{lock}) \neq \mathsf{thread}$ if p is the precondition of a call invoking a monitor method of e_0,

- $\exists \vec{v}.\, p[z/\mathsf{thread}][z'/\mathsf{this}] \wedge (z'.\mathsf{lock} \neq \mathsf{free} \vee z \notin z'.\mathsf{notified})$ if p is the precondition of the return statement in the wait method, and

- false, otherwise,

where $\vec{v}$ is the vector of local variables in p, and z and z' are fresh. Note that thread is substituted by a logical variable and thus the quantification over thread is without effect. Let finally $\mathsf{blocked}(z, z')$ express that the thread of object z is blocked in the object z'. It is defined by the assertion $\bigvee_{p \in Ass(z')} \mathsf{blocked}(z, z', p)$, where $Ass(z')$ is the set of all assertions in z'. Now we can formalize the verification condition for deadlock freedom:

Definition 7.1.1 *A proof outline satisfies the test for* deadlock freedom, *if*

$$
\begin{aligned}
\models_{\mathcal{G}} \quad & (GI \wedge \\
& (\forall z.\, z \neq \mathsf{null} \rightarrow (I(z) \wedge \\
& \quad (z.\mathsf{started} \rightarrow (\mathsf{terminated}(z) \vee (\exists z'.\, z' \neq \mathsf{null} \wedge \mathsf{blocked}(z, z')))))) \wedge \\
& (\exists z.\, z \neq \mathsf{null} \wedge z.\mathsf{started} \wedge (\exists z'.\, z' \neq \mathsf{null} \wedge \mathsf{blocked}(z, z')))) \\
& \rightarrow \mathsf{false}\ .
\end{aligned}
\tag{7.1}
$$

Soundness of the above condition, i.e., that the condition indeed assures absence of deadlock, is easy to show. Relative completeness results from the relative completeness of the proof method.

7.2 Examples of proofs of deadlock freedom

Next we illustrate the application of the proof system to show absence of deadlock on some examples. All examples are verified using *PVS*.

For readability, we define the following functions, which describe properties of synchronization:

$$\begin{aligned}
\mathit{owns} \quad &: \quad (\mathsf{Thread} \times (\mathsf{Thread} \times \mathsf{Int})) \rightarrow \mathsf{Bool}, \\
&\quad\ \mathit{owns}(\mathit{thread}, \mathit{lock}) \stackrel{\mathit{def}}{=} \mathit{thread} \neq \mathit{null} \wedge \mathit{proj}(\mathit{lock}, 1) = \mathit{thread} \\
\mathit{not_owns} \quad &: \quad (\mathsf{Thread} \times (\mathsf{Thread} \times \mathsf{Int})) \rightarrow \mathsf{Bool}, \\
&\quad\ \mathit{not_owns}(\mathit{thread}, \mathit{lock}) \stackrel{\mathit{def}}{=} \mathit{thread} \neq \mathit{null} \wedge \mathit{proj}(\mathit{lock}, 1) \neq \mathit{thread} \\
\mathit{depth} \quad &: \quad (\mathsf{Thread} \times \mathsf{Int}) \rightarrow \mathsf{Int}, \\
&\quad\ \mathit{depth}(\mathit{lock}) \stackrel{\mathit{def}}{=} \mathit{proj}(\mathit{lock}, 2)\ .
\end{aligned}$$

The function *proj* is defined in Lemma 6.1.2; the *owns* function is already used in Example 4.3.2. In the following we apply the test for deadlock freedom to some examples. The built-in augmentation is not listed in the code. Again, we additionally list instance and local variable declarations `type name;`, where $\langle$`type name;`$\rangle$ declares auxiliary variables. We sometimes skip return statements without giving back a value, and write explicitly $\forall(z:t).p$ for quantification over t-typed values. All missing assertions are by definition true. An empty auxiliary observation $\langle\rangle$ in a notify or notifyAll method represents the built-in auxiliary assignment in the given method.

7.2.1 Reentrant monitors

To demonstrate the basic idea of proving absence of deadlock, we first define a simple program, which does the following: The initial object, an instance of class `Main`, creates an instance of class `Synch`, starts its thread, and calls its synchronized `m1` method. The thread of the created instance also invokes `m1`, which simply calls the synchronized method `m2` of itself. Since synchronized methods cannot be executed simultaneously by different threads[1], either the initial thread or the thread of the new object calls `m1`, and then `m2`. The other thread has to wait until control returns from `m1`, before it can execute the invocations. The program is deadlock free, since *Java*'s monitor concept is reentrant, i.e., a thread owning the lock of an object may invoke several synchronized methods of that object.

Appendix B.1 contains a proof outline which satisfies the verification conditions and which implies the following invariant program properties:

```
class Main{
    ⟨ Bool in_Synch; ⟩
    ⟨ Synch created; ⟩

    nsync Void wait(){ {false} }

    nsync Void run(){
        Synch obj;
        obj := new^Synch; ⟨created := obj⟩^new
        obj.start();
```

[1]if non of them is in the *wait* or *notified* set of the given object

```
            {(¬in_Synch) ∧ created = obj ∧ thread = this ∧ obj ≠ null ∧ obj ≠ this}
            obj.m1()  ⟨in_Synch := (if obj = this then in_Synch else true fi)⟩^!call
                      ⟨in_Synch := (if obj = this then in_Synch else false fi)⟩^?ret
            {¬in_Synch}
      }
}

class Synch{

      nsync Void wait(){ {false} }

      sync Void m1(){
            {owns(thread, lock)}
            m2()
      }

      sync Void m2(){}

      nsync Void run(){
            {not_owns(thread, lock) ∧ thread = this ∧ started}
            m1()
            {not_owns(thread, lock)}
      }
}
```

with global invariant

$$
\begin{aligned}
& GI \stackrel{def}{=} \\
& (\forall(z : Synch).\, z \neq null \rightarrow (z.lock = (null, 0) \vee \\
& \qquad (\exists(t : Main).\, owns(t, z.lock) \wedge t.started \wedge t.created = z) \vee \\
& \qquad (owns(z, z.lock) \wedge z.started))) \wedge \\
& (\forall(t : Main).\, (t \neq null \wedge (\neg t.in\_Synch)) \rightarrow (t.created = null \vee not\_owns(t, t.created.lock))) \wedge \\
& (\forall(t : Main).\, t \neq null \rightarrow (\forall(z : Synch).\, (z \neq null \wedge owns(t, z.lock)) \rightarrow t.created = z))\,.
\end{aligned}
$$

The annotation shows properties at control points with terminated or possibly disabled execution, and implies that a disabled or terminated thread owns the lock of a `Synch`-instance only if its current control point resides in a synchronized method of the object. For threads of `Main`-instances this property cannot be expressed locally, thus we use the boolean auxiliary instance variable `in_Synch` to remember if the control point of the thread of the `Main`-instance is in itself or in the `Synch`-instance `obj`. To be able to refer to the identity of `obj` in the global language, we store the same identity in the auxiliary instance variable `created`. The global invariant GI combines properties of `Main`- and `Synch`-instances, stating that the lock of `Synch`-instances is either free or owned by the creator of the instance or by the instance itself. Furthermore, if the variable `in_Synch` of a `Main`-instance z has the value $false$, then the thread of z does not hold the lock of z.created; `Main`-instances can own only the lock of the `Synch`-instance which they have been created.

The left-hand-side of the implication in the deadlock freedom condition states that there is an object $z \neq \mathsf{null}$ whose thread is already started and whose execution is disabled in another object $z' \neq \mathsf{null}$, i.e., $\mathsf{blocked}(z, z')$. First assume that z' refers to a `Main`-instance. Then the assertion $\mathsf{blocked}(z, z')$ implies that $z = z'$ is of type `Main`, and the thread of z tries to invoke method `m1` of z'.created with

$$z'.\mathsf{created} \neq \mathsf{null}\,, \tag{7.2}$$

where the lock of $z'.\mathsf{created}$ is neither free nor owned by z, and $\neg z'.\mathsf{in\_Synch}$ holds. Using the global invariant we obtain that there is an already started thread which owns the lock of $z'.\mathsf{created}$.

The antecedent of the deadlock freedom condition assures furthermore that the execution of the lock owner is either disabled or terminated. Let the current control point of the lock owner reside in an object z''. This object cannot be a `Main`-instance: The assertions at both possible control points imply that the executing thread is the thread of z'' and that $\neg z''.\mathsf{in\_Synch}$ holds. Using the global invariant we obtain on the one hand

$$z''.\mathsf{created} = \mathsf{null} \vee \mathsf{not\_owns}(z'', z''.\mathsf{created.lock})\,, \tag{7.3}$$

and on the other hand GI states that the lock of $z'.\mathsf{created}$ can be owned by the object itself or by its creator, i.e., the assumption $\mathsf{owns}(z'', z'.\mathsf{created.lock})$ implies $z''.\mathsf{created} = z'.\mathsf{created}$, i.e.,

$$\mathsf{owns}(z'', z'.\mathsf{created.lock}) \wedge z''.\mathsf{created} = z'.\mathsf{created}\,. \tag{7.4}$$

Note that 7.2, 7.3, and 7.4 together lead to a contradiction. Thus the lock owner executes in a `Synch`-instance. We have three possible control points of the lock owner:

- The first possibility, prior to the invocation of `m2` in `m1` of z'', directly leads to a contradiction by the definition of the assertion $\mathsf{blocked}$: The precondition of the invocation states that the thread does own the lock of z'', and $\mathsf{blocked}$ extends this assertion by the assumption that the execution is not enabled, i.e., that the thread does not own the given lock.

- In the second case the lock owner is about to invoke `m1` in the `run` method of z''. From the precondition of the invocation we get that the executing thread is the thread of z''. The global invariant implies that `Synch`-instances cannot own the lock of other `Synch`-instances. Now, by assumption z'' owns the lock of $z'.\mathsf{created}$, and with the above observation we obtain that $z'' = z'.\mathsf{created}$, i.e., z'' owns its own lock. But the precondition of the invocation implies that the thread does not own the lock of z'', which leads to a contradiction.

- In the third case, the lock owner is the thread of z'' and is terminated. Again, the assumption that the executing thread, i.e., z'', owns the lock of $z'.\mathsf{created}$ implies with GI that $z'' = z'.\mathsf{created}$, i.e., that z'' owns its own lock. But the assertion at the given control point implies that z'' does not own its own lock, which leads again to a contradiction.

For the case that z' refers to a `Synch`-instance, we obtain from $\mathsf{blocked}(z, z')$ that the lock of z' is not free, but z is not the owner. The global invariant implies again that there is an object whose thread is started and owns the lock of z'. The rest is analogous to the above case, where $z'.\mathsf{created}$ is replaced by z'.

7.2.2 A simple wait-notify example

Now let's have a look at an example demonstrating deadlock freedom for a notification process. Assume a program which defines two classes: The initial instance of the main class `Main` creates an instance of the class `Monitor`, and invokes its synchronized method `m1`, which starts its thread, and suspends the executing thread, thereby giving the lock free. Now the thread of the `Monitor`-instance can execute the synchronized method `m2`, probably producing some results which the other thread is waiting for. After the computation is completed, the lock owner sends a notification, and returns from `m2`. Now the other thread can continue its execution and use the produced data.

Again, Appendix B.2 lists a proof outline, which satisfies the verification conditions, and which implies the following invariant program properties:

$$
\begin{array}{l}
GI \overset{def}{=} \\
\quad (\forall(z_1, z_2 : Main).(z_1 \neq null \wedge z_2 \neq null) \rightarrow z_1 = z_2) \wedge \\
\quad (\forall(z_1, z_2 : Monitor).(z_1 \neq null \wedge z_2 \neq null) \rightarrow z_1 = z_2) \wedge \\
\quad (\forall(z : Main).z \neq null \rightarrow (\\
\qquad z.started \wedge \\
\qquad (z.x = 1 \rightarrow (z.created \neq null \wedge z.created.lock = (null, 0))) \wedge \\
\qquad (z.x = 3 \rightarrow (z.created \neq null \wedge z.created.x = 8)))) \wedge \\
\quad (\forall(z_1 : Main).z_1 \neq null \rightarrow \\
\qquad (\forall(z_2 : Monitor).(z_2 \neq null \wedge owns(z_1, z_2.lock)) \rightarrow z_2 = z_1.created)) \wedge \\
\quad (\forall(z_1 : Monitor).z_1 \neq null \rightarrow \\
\qquad (\forall(z_2 : Monitor).(z_2 \neq null \wedge owns(z_1, z_2.lock)) \rightarrow (z_1.started \wedge z_2 = z_1)))
\end{array}
$$

$$
\begin{array}{l}
I_{Monitor} \overset{def}{=} \\
\quad ((x = 2 \vee x = 7) \rightarrow (lock = (creator, 1) \wedge started)) \wedge \\
\quad ((x = 4 \vee x = 5) \rightarrow (lock = (this, 1) \wedge started)) \wedge \\
\quad (x = 6 \rightarrow (lock = (null, 0) \wedge creator \in notified \wedge started))) \wedge \\
\quad ((x = 3 \vee x = 8) \rightarrow lock = (null, 0) \wedge started)
\end{array}
$$

```
class Main{
    ⟨ Int x; ⟩
    ⟨ Monitor created; ⟩

    nsync Void wait(){ {false} }

    nsync Void run(){
        Monitor obj;
        obj := new^Monitor; ⟨created, x := obj, 1⟩^new
        {x = 1 ∧ thread = this ∧ created = obj ∧ obj ≠ null}
        obj.m1() ⟨x := (if obj = this then x else 2 fi)⟩^!call
                 ⟨x := (if obj = this then x else 3 fi)⟩^?ret
        {x = 3}
    }
}

class Monitor{
    ⟨ Main creator; ⟩
    ⟨ Int x; ⟩

    nsync Void wait(){
        ⟨x := 3⟩^?call
        {3 ≤ x ∧ x ≤ 6 ∧ thread = creator}
        return_getlock ⟨x := 7⟩^!ret

    }
```

```
    nsync Void notify(){ ⟨⟩ return ⟨x := 5⟩^!ret }

    sync Void m1(){
        ⟨creator, x := thread, 1⟩^?call
        start();
        {x = 2 ∧ thread = creator}
        wait();
        return ⟨x := 8⟩^!ret
    }

    nsync Void run(){
        ⟨x := 2⟩^?call
        {(x = 2 ∨ x = 3) ∧ thread = this}
        m2()
        {x = 6 ∨ x = 7 ∨ x = 8}
    }

    sync Void m2(){
        ⟨x := 4⟩^?call
        {x = 4 ∧ thread = this}
        notify();
        return ⟨x := 6⟩^!ret
    }
}
```

Note that the precondition of the method invocation in the `run` method of `Main` together with the global invariant implies that the lock of the callee is free, i.e., threads cannot be blocked at this control point. Furthermore, the preconditions of both monitor method calls in `Monitor` imply with the class invariant that the executing thread owns the lock, i.e., also at these control points execution is always enabled.

We start again with the assumption that there is an object z whose thread is started but not yet terminated, and whose execution is disabled in the object z', where the values of both z and z' are different from the empty reference. The object z can be an instance of one of the classes `Main` or `Monitor`. According to the above observations, z' must be an instance of `Monitor`, and the control point is in the `wait` method or prior to the invocation of `m2` in the `run` method.

In the first case, the local assertion attached to the control point in the `wait` method implies that $z = z'.\mathsf{creator}$, an instance of `Main`, does not own the lock of z' and that the thread of z' is started. Due to the assumptions of the deadlock freedom condition, the execution of the thread of z' is disabled or terminated. However, using the annotation, termination would imply $z'.x = 6$ and by the class invariant the execution of the thread of z would be enabled. The thread of z' can neither be in the `wait` method, because the local assertion there implying $\mathsf{thread} = \mathsf{creator}$ would lead to a type contradiction. Thus the thread of z' executes the `run` method of z', and is going to invoke the synchronized method `m2`. Since $z = z'.\mathsf{creator}$ does not own the lock of z' by assumption, the precondition of the invocation and the class invariant imply that the lock is free, and thus that the execution of z' is enabled.

The second case, when the thread of z resides in the `run` method of z' prior to the call of `m2`, is similar.

7.2.3 A producer-consumer example

The proof outline below defines two classes `Producer` and `Consumer`, where `Producer` is the main class. The initial thread of the initial `Producer`-instance creates a `Consumer`-instance and calls its synchronized `produce` method. This method starts the consumer thread and enters a non-terminating loop, producing some results, notifying the consumer, and suspending itself by calling `wait`. After the producer suspended itself, the consumer thread calls the synchronized `consume` method, which consumes the result of the producer, notifies, and calls `wait`, again in a non-terminating loop.

Again, we only list a partial annotation and augmentation, which already implies deadlock freedom; see Appendix B.3 for the complete inductive proof outline.

$$
\begin{aligned}
&GI \stackrel{def}{=}\\
&\quad (\forall(p : Producer).(p \neq null \wedge \neg p.outside \wedge p.consumer \neq null) \rightarrow\\
&\qquad p.consumer.lock = (null, 0))\wedge\\
&\quad (\forall(c : Consumer).(c \neq null \wedge c.started) \rightarrow (c.producer \neq null \wedge c.producer.started))\wedge\\
&\quad (\forall(c1 : Consumer).(c1 \neq null \rightarrow (\forall(c2 : Consumer).c2 \neq null \rightarrow c1 = c2))
\end{aligned}
$$

$$
\begin{aligned}
&I_{Producer} \stackrel{def}{=} true\\
&I_{Consumer} \stackrel{def}{=} length(wait) \leq 1\wedge\\
&\quad (lock = (null, 0) \vee (owns(this, lock) \wedge started) \vee owns(producer, lock))
\end{aligned}
$$

```
class Producer {
    ⟨ Consumer consumer; ⟩
    ⟨ Bool outside; ⟩

    nsync Void wait(){ {false} }

    nsync Void run(){
        Consumer c;
        c := new^Consumer; ⟨consumer := c⟩^new
        {c = consumer ∧ ¬outside ∧ consumer ≠ null ∧ consumer ≠ this ∧ thread = this}
        c.produce() ⟨outside := (if c = this then outside else true fi)⟩^!call
        {false}
    }
}

class Consumer {
    Int buffer;
    ⟨ Producer producer; ⟩

    nsync Void wait(){
        {started ∧ not_owns(thread, lock) ∧ (thread = this ∨ thread = producer)∧
            (thread ∈ wait ∨ thread ∈ notified)}
    }

    sync Void produce(){
        Int i;

        ⟨producer := proj(caller, 1)⟩^?call
        i:=0;
        start();
        while (true) do
            //produce i here
            buffer := i;
            {owns(thread, lock)}
            notify();
```

```
            {owns(thread, lock)}
            wait()
        od
    }

    nsync Void run(){
        {not_owns(thread, lock) ∧ thread = this}
        consume()
        {false}
    }

    sync Void consume(){
        Int i;

        while (true) do
            i := buffer;
            //consume i here
            {owns(thread, lock)}
            notify();
            {owns(thread, lock)}
            wait()
        od
    }
}
```

Both `run` methods have false as postcondition, stating that the corresponding threads do not terminate. The preconditions of all monitor method invocations express that the executing thread owns the lock, and thus execution cannot be enabled at these control points. The `wait` method of `Producer`-instances is not invoked; we define false as the precondition of its return-statement, implying that disabledness is excluded also at this control point.

The condition for deadlock freedom assumes that there is a thread which is started but not yet terminated, and whose execution is disabled. This thread is either the thread of a `Producer`-instance or that of a `Consumer`-instance.

We discuss only the case that the disabled thread belongs to a `Producer`-instance z different from the empty reference; the other case is similar. Note that the control of the thread of z cannot stay in the `run` method of a `Consumer`-instance, since the corresponding local assertion implies thread = this, which would contradict to the type assumptions. Thus the thread can have its control point prior to the method call in the `run` method of a `Producer`-instance, or in the `wait` method of a `Consumer`-instance. In the first case, the corresponding local assertion and the global invariant imply that the lock of the callee is free, i.e., that the execution is enabled, which leads to a contradiction. In the second case, if the thread of z executes in the `wait` method of a `Consumer`-instance z', the local assertion in `wait` together with the type assumptions implies z'.started $\wedge$ not_owns(z, z'.lock) $\wedge\, z = z'$.producer, and that z is either in the wait or in the notified set of z'.

According to the assumptions of the deadlock freedom condition, also the started thread of z' is disabled or terminated; its control point cannot be in a `Producer`-instance, since that would contradict to the type assumptions. Thus the control of z' stays in the `run` or in the `wait` method of a `Consumer`-instance; the annotation implies that the instance is z' itself.

If the control stays in the `run` method, then the corresponding local assertion and the class invariant imply that the lock is free, since neither the producer, nor the consumer owns it, which leads to a contradiction, since in this case the execution of the thread of z' would be enabled. Finally, if the control of the thread of z' stays in the `wait` method of z', then the annotation assures that the thread does not own the lock of z'; again, using the class invariant we get that the lock is free.

Now, both threads of z and z' have their control points in the `wait` method of z', and the lock of z' is free. Furthermore, both threads are disabled, and are in the wait or in the notified set. If one of them is in the notified set, then its execution is enabled, which is a contradiction. If both threads are in the wait set, then from $z \neq z'$ we imply that the wait set of z' has at least two elements, which contradicts the class invariant of z'.

Thus the assumptions lead to a contradiction, which was to be shown.

7.3 Conclusions and related work

This chapter introduced a verification condition for establishing deadlock freedom and illustrated its use on some examples.

There are just a few works on proof systems for establishing deadlock freedom of *Java* programs.

Demartini et al. [DIS98, IDS99] describe how core features of multithreaded *Java* can be mapped into the Promela language of the SPIN model checker to prove deadlock freedom.

Boyapati et al. [BSBR03] present a static type system for multithreaded programs. Well-typed programs are guaranteed to be free of deadlock. However, they only take synchronization via object locks into account, but no wait-notify constructs.

Chapter 8

Possible extensions

In the previous sections we introduced proof systems for three languages, defined incrementally by extensions. Though we used an abstract syntax for readability, the languages can be seen as *Java* sublanguages; tool support is developed for *Java* syntax. Besides some semantical differences between Java and our languages, in this chapter we discuss a number of possible further extensions of the language and the proof system.

8.1 *Java*'s memory model

In our language we assume a global state. Each expression gets evaluated in this global state, and the execution of assignments affects directly the global state.

In contrast, in *Java*'s memory model every thread has a *working memory*, in which it keeps its own *working copy* of variables that it uses or assigns. Threads operate on these working copies. The *main memory* contains the *master copy* of every variable. From [GJSB00]: " There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy and vice versa. [...] If the implementation correctly follows these rules and the application programmer follows certain other rules of programming, then data can be reliably transferred between threads through shared variables. The rules are designed to be 'tight' enough to make this possible but 'loose' enough to allow hardware and software designers considerable freedom to improve speed and throughput through such mechanisms as registers, queues, and caches." For a detailed description of the *Java* memory model we refer to [GJSB00]. Here we only briefly describe a part of the model and demonstrate it on a small example.

We call operations on memories *actions.* A *read* action by the main memory transmits the contents of the master copy of a variable to a thread's working memory for use by a later *load* action. A *load* action of a thread puts a value transmitted from the main memory by a *read* action into the thread's working copy of a variable. Whenever a thread executes a virtual machine instruction that uses the value of a variable, it transfers the contents of the thread's working copy of the variable to the thread's execution engine. We use the name *use* for this action.

An *assign* action of a thread is performed whenever a thread executes a virtual machine instruction that assigns to a variable, and it transfers a value from the thread's execution engine into the thread's working copy of a variable. A *store* action of a thread transmits the contents of the thread's working copy of a variable to the main memory for use by a later *write* action. A *write* action by the main memory puts a value transmitted from a thread's working memory by a *store* action into the master copy of a variable in the main memory.

Thus threads operate on variables by *use*, *assign*, *load*, and *store* actions, where the main memory performs a *read* action for every *load* and a *write* action for every *store*. The order of the above actions is not strongly coupled. For example, under some restrictions it is possible that a *store* action happens before the corresponding *assign* action: In this case the *store* action sends to the main memory the value that the *assign* action will put into the working memory of the executing thread. This is called a *prescient* store action.

Additionally, a thread's interaction with a lock of an object over time consists of a sequence of *lock* and *unlock* actions. A *lock* action acts as if it flushes all variables from the thread's working memory; before use they must be assigned to or loaded from the main memory. If a thread performs an *unlock* action on any lock, it must first copy all assigned values in its working memory back to the main memory.

The guarantees made by the memory model are weaker than most programmers intuitively expect, and are also weaker than those typically provided on any *JVM* implementation. Locking objects before accessing any instance variables guarantee that values are correctly transmitted from one thread to another through shared variables. Note that locking any lock flushes *all* variables from the thread's working memory, and unlocking any lock forces the writing out of *all* variables that the thread has assigned into the main memory. That the lock is associated with an object does not play any role in this context.

The following example demonstrates how the complexities arising in concurrent *Java* programs can be avoided using synchronization.

Example 8.1.1 *This example is a modification of an example from [GJSB00]. Assume two assignments $x := y$ and $y := x$ to instance variables x and y, executed by the threads t_1 and t_2, respectively, in the same object in a state satisfying $x = 1$ and $y = 2$.*

What is the required set of actions and what are the ordering constraints? Execution of t_1 causes the following actions[1], where the read and the write actions are by the main memory, and the remaining ones by the thread, and where

[1] The implementation may also choose not to perform the *store* and *write* actions, or only one of the two pairs for t_1 and t_2, leading to further possible results.

an arrow from action A to action B indicates that A must precede B:

$$\boxed{\textit{read } y} \rightarrow \textit{load } y \rightarrow \textit{use } y \rightarrow \textit{assign } x \rightarrow \textit{store } x \rightarrow \boxed{\textit{write } x}$$

For the thread t_2 we have similarly:

$$\boxed{\textit{read } x} \rightarrow \textit{load } x \rightarrow \textit{use } x \rightarrow \textit{assign } y \rightarrow \textit{store } y \rightarrow \boxed{\textit{write } y}$$

The only constraint on the order of the main memory actions is that not both write actions precede both read actions. Let x_i and y_i denote the working copies of x and y for the thread t_i, $i = 1, 2$. The three possible orderings of the main memory actions and the resulting states are given by

$$\begin{aligned}
&\textit{if write } x \rightarrow \textit{read } x \textit{ and read } y \rightarrow \textit{write } y \\
&\qquad\qquad \textit{then } x = 2,\ y = 2,\ x_1 = 2,\ y_1 = 2,\ x_2 = 2,\ y_2 = 2 \\
&\textit{if read } x \rightarrow \textit{write } x \textit{ and write } y \rightarrow \textit{read } y \\
&\qquad\qquad \textit{then } x = 1,\ y = 1,\ x_1 = 1,\ y_1 = 1,\ x_2 = 1,\ y_2 = 1 \\
&\textit{if read } x \rightarrow \textit{write } x \textit{ and read } y \rightarrow \textit{write } y \\
&\qquad\qquad \textit{then } x = 2,\ y = 1,\ x_1 = 2,\ y_1 = 2,\ x_2 = 1,\ y_2 = 1
\end{aligned}$$

That means, either the value of y is copied into x, or the value of x is copied into y, or the values of x and y are swapped; moreover, the working copies of the variables might or might not agree.

Now we modify the example by assuming that both assignments represent the body of synchronized methods of the same object. In this case both threads must perform a lock action on the object before execution, and an unlock action on the same instance after the body of the method completes. These actions provide further constraints on the ordering: The lock action of one of the threads cannot occur between the lock and unlock actions of the other thread. Moreover, after the lock actions all used variables must be assigned to or loaded from the main memory (since the lock action flushes all variables from the working memory); the unlock actions require that the store and write actions occur, i.e., all assigned values must be copied from the working memory into the main memory before the lock gets released. Thus the actions of t_1 are:

$$\boxed{\textit{lock}} \rightarrow \boxed{\textit{read } y} \rightarrow \textit{load } y \rightarrow \textit{use } y \rightarrow \textit{assign } x \rightarrow \textit{store } x \rightarrow \boxed{\textit{write } x} \rightarrow \boxed{\textit{unlock}}$$

For the thread t_2 we have similarly:

$$\boxed{\textit{lock}} \rightarrow \boxed{\textit{read } x} \rightarrow \textit{load } x \rightarrow \textit{use } x \rightarrow \textit{assign } y \rightarrow \textit{store } y \rightarrow \boxed{\textit{write } y} \rightarrow \boxed{\textit{unlock}}$$

It follows that we have only two possible sequences of the main memory actions read and write:

$$\begin{aligned}
&\textit{if write } x \rightarrow \textit{read } x \textit{ and read } y \rightarrow \textit{write } y \\
&\qquad\qquad \textit{then } x = 2,\ y = 2,\ x_1 = 2,\ y_1 = 2,\ x_2 = 2,\ y_2 = 2 \\
&\textit{if read } x \rightarrow \textit{write } x \textit{ and write } y \rightarrow \textit{read } y \\
&\qquad\qquad \textit{then } x = 1,\ y = 1,\ x_1 = 1,\ y_1 = 1,\ x_2 = 1,\ y_2 = 1
\end{aligned}$$

The threads necessarily agree on the values of x and y; they cannot be swapped.

Our assertional proof system for the concurrent *Java*$_{synch}$ language basing on a global state is already complex. Though it would be possible to formalize an assertional proof system taking *Java*'s real memory model into account, its complexity would be enormous. The better way would be to avoid or reduce somehow these complexities.

As we have seen, a simple solution is given by synchronization. Of course, allowing exclusively synchronized methods —or at least assuming all methods containing references to instance variables to be synchronized— would be a very strong restriction on concurrency. So let us search for further solutions.

Instead of making all computations involving instance variables mutually exclusive, we make a less restrictive requirement, namely that all access or update to instance variables can be seen as atomic. A possibility to implement this requirement is offered by the *Java* modifier *volatile*. Declaring an instance variable as volatile is nearly identical in effect to using a little fully synchronized class protecting only that instance variable via get/set methods; it differs only in that no locking is involved, but it assures mutually exclusive access or updates to the volatile instance variable itself. Declaring volatile instance variables seems to be cheaper than synchronization at the first site. However, frequent access to volatile instance variables leads to slower performance then locking. Another solution is to implement synchronized get and set methods for all instance variables, and access them only through the invocation of these methods. If synchronization using the object's lock is disadvantageous, the same effect can be reached by creating for each object another object whose lock can be used for synchronization for instance variable access.

The semantics of the previous chapters define assignment, object creation, the invocation of a method, and returning from a method to be atomic, i.e., to be executed in one computation step without interleaving. In the following we discuss how to modify the proof system if only instance variable access and update can be assumed to be atomic.

Assume a thread t executing the annotated assignment $x := y$ $\{x = y\}$, where x and y are instance variables. This execution could interleave with other threads, such that first t reads y, then another thread changes the value of y by executing $\{u \neq y\}$ $y := u$ with local variable u, and then t assigns the old value of y to x. As a consequence, it is possible that after the execution of the assignment $x := y$ the assertion $x = y$ is not satisfied.

At first sight one might have the impression that the proof system does not handle this interleaving case. But it does so: The interference freedom test condition requiring invariance of $x = y$ under $\{u \neq y\}$ $y := u$ fails.

Unfortunately, it does not work in cases where both assignments contain instance variables on both their right- and left-hand sides, i.e., if both assignments read and write instance variable values, like it is the case in Example 8.1.1. The assignments $x := y$ $\{x = y\}$ and $y := x$ $\{x = y\}$ of the example executed by the threads t_1 and t_2, respectively, in the same object in a state satisfying $x = 1$ and $y = 2$, can be interleaved as follows: t_1 reads y having the value 2, t_2 executes $y := x$ resulting in $x = y = 1$, and, finally, t_1 sets the value of x to the old value of y leading to $x = 2$ and $y = 1$. Since the proof system is based on a semantics with atomic assignments, the annotation satisfies the verification conditions. However, real *Java* semantics can lead to a state where the annotation is not satisfied.

There are different solutions to overcome this semantical difference: First, we could change the semantics and extend the annotation by attaching additional assertions to each real *Java* control point. For the above example, besides the annotation $\{p_1\}\ x := y\ \{p_3\}$ we could define an additional assertion p_2 which should additionally hold directly after storing the value of y in an auxiliary local variable u_y (representing the working copy of y) and before writing x. Local correctness should require that p_1 implies p_2 after assigning the value of y to u_y, and that p_2 implies p_3 after assigning the value of u_y to x. The assertion p_2 should be included into the interference freedom test, to assure its invariance under execution. Note that we would need such an additional assertion for each occurrence of instance variables on the right-hand side of assignments, which should hold directly after reading the value of the given instance variables.

Another —perhaps more natural— solution would be to replace the assignment $x := y$ by $u := y; x := u$, where u is a local variable. In this case, if there is at most one occurrence of instance variables per assignment, we can consider assignments as atomic. The control point between $u := y$ and $x := u$ is the one where p_2 from above should hold after reading but prior to writing. Due to this replacement, the proof system assures inductivity without further modifications.

For communication and object creation we do not have such semantical problems, since we do not allow the occurrence of instance variables in such statements.

But anyhow, the best solution is to use proper synchronization in *Java* programs which assures that the program does what one would expect. If a *Java* program does not satisfy the assumption that assignments which do not contain side-effect expressions on the right-hand side are executed without interleaving, then even for the programmer it will not be clear what the program does. Our interleaving abstraction is a natural one; if a program does not satisfy our requirements than probably the best solution is to reformulate the program.

8.2 Weakening the language restrictions

In our languages we made some syntactical restrictions. One of these restrictions is that for self-communication the caller observation may not change the instance state. The reason for introducing this restriction is to simplify the interference freedom test: If both the caller and the callee observation in a self-communication modify the instance state, then we have to show invariance not only under multiple assignments, but also under assignment pairs, since caller and callee observations are executed in a single computation step. We can release this restriction by modifying the formulation of the interference freedom test as follows: The condition for invariance of assertions under assignments which do not observe communication does not change. For observations of communication we use the same condition under the assumption that the call is not a self-call, which can be expressed using the built-in augmentation. We need an additional interference freedom condition for invariance of assertions under the assignment-pairs of caller *and* callee observations for self-calls. Similar conditions apply for the return case.

A further restriction is that the results of communication and object creation must not be assigned to instance variables. This restriction could be released without loosing the modularity of the proof system, i.e., while still keeping separate tests for interference freedom and cooperation. To do so, we should separately handle communication itself and the assignment of the result to the instance variable. Assume a (partially) annotated method call statement $\{p_1\}\ x := e_0.m(\vec{e})\ \{p_2\}^{wait}\,\{p_3\}^{?ret}\,\langle stm\rangle^{?ret}\,\{p_4\}$. We introduce an additional auxiliary local variable u_x, which allows us to refer to the return value in the assertion p_3, which should hold directly after communication but *before* assigning the return value to x. The observation *stm* should additionally contain the assignment $x := u_x$ representing the storage of the result. The cooperation test for the return case gets modified in that the substitution representing communication replaces u_x by the return expression, instead of replacing x. The other conditions do not change.

We could similarly release the restriction that actual parameters may not contain instance variables. We used this restriction to assure that the values of the actual parameters are not modified during method evaluation, and thus the actual parameter expressions can be used to express caller-callee relationship also for returning from the method. If we do allow references to instance variables in actual parameter expressions, then we have to store the actual parameter values at method invocation in additional auxiliary local variables, so that we can refer to the actual parameter values in the condition for the return case. We extend the observation of the caller for a method call by assigning the actual parameter expressions to those auxiliary variables. Only the cooperation test for the return case changes, where the actual parameter expressions get replaced by the auxiliary variables storing the actual parameter values at method call.

We can similarly allow that the expression e_0 specifying the callee object in method call statements $e_0.m(\vec{e})$ contains instance variables. As for the actual parameters, we need to store the callee identity in an additional auxiliary local variable, and modify the cooperation test for the return case correspondingly in order to refer to the auxiliary variable storing the callee identity instead of referring to e_0.

Also formal parameters could be assigned to, if we store their values at method call in special auxiliary local variables, and use those variables to express caller-callee relationship in the cooperation test for the return case.

Another restriction we made on the language is that variables occurring in the global invariant may be assigned to only in the observations of object creation and communication. In other words, the global invariant is meant to express properties of object creation and communication, i.e., referring to inter-object behavior, only, but not intra-object properties. Our experience during the application of the proof system to some examples has shown that this restriction sometimes increases the complexity of the augmentation and annotation, because we additionally have to express dependencies between inter- and intra-object behavior in class invariants in order to combine properties expressed in the global invariant with those formulated in local assertions.

Also this restriction can be released. If we allow the assignment to variables occurring in the global invariant also outside of observations of communication and object creation, we have to extend the cooperation test with a condition, which assures invariance of the global invariant under such assignments. This condition should state that if the global invariant, the class invariant of the object in which the execution takes place, and the precondition of the assignment hold, then the global invariant holds after the execution of the assignment.

8.3 Constructors

Constructors allow one to execute some statements directly after the creation of an object, leading to the new object's user-defined initialization. Constructors can be handled simply by treating object creation in two steps: the creation itself together with a method invocation calling the constructor method. If constructor methods may contain also communication statements, interference freedom must apply also to their statements and assertions.

We could also restrict the usage of constructors by requiring that they do not contain communication or object creation statements. In this case no interleaving can take place in the new object during the execution of the constructor method, and thus we wouldn't have to apply the interference freedom test to constructor methods.

8.4 Static variables and methods

Static variables and methods belong to classes instead of objects, and exist during the whole program execution. They can be represented by special objects, one for each class, containing the static variables and methods of the class. These special objects are already included into the initial configuration, and no new instances of their types can be created, i.e, their existence is static and not dynamic like the existence of objects. In these objects no new threads can be started. All verification conditions would apply also to the static constructs.

8.5 Exceptions

The extension of the programming language and the proof system with exception handling (without inheritance) is underway [ÁdBdRS04b]. It is a straightforward translation of the transition rules of the operational semantics into verification conditions, as demonstrated for other language constructs during the incremental development of the proof system for the concurrent language with monitor synchronization.

8.6 Inheritance

The extension of the proof system to cover inheritance requires more effort. Dealing with subtyping on the logical level requires a notion of behavioral subtyping [Ame89]. The work [PdB03] introducing an assertional proof system for a sequential language covering inheritance might provide a basis for a similar proof system for our concurrent language. This should be possible, since concurrency and inheritance are "orthogonal" in a proof-theoretical sense. Of course, we additionally have to cover the effect of interleaving.

Chapter 9

Tool support

We see the formulation of a sound and complete proof system, providing a logical and modular characterization of the concurrency aspects of *Java*, not only as interesting in itself. The usage of the proof rules like the ones presented here in actual verification needs a reasonable amount of mechanized tool support. The theory presented in the previous sections forms, therefore, the theoretical foundation for the verification tool *Verger* which takes *Java* programs asserted in an adaptation of *JML* notation and generates verification conditions for the *PVS* theorem prover. The conditions can be verified interactively using *PVS*.

Most of the examples of this chapter are already treated in the previous chapters. Here we reformulate the examples in *Java* syntax and discuss the syntactical verification conditions generated by *Verger*.

9.1 The theorem prover *PVS*

Theorem provers offer mechanized support for logical reasoning in general and for program verification in particular. Unlike verification systems for fully automated reasoning such as model checkers [CGP99], theorem provers provide machine-*assistance,* i.e., an interactive proof environment. Interactive means that the user is requested to organize the proof, for instance to come up with an induction hypothesis, to split the proof in appropriate lemmas, etc. While doing so, the verification environment takes care of tedious details like matching and unifying lemmas with the proof goals and assists in the proof organization by keeping track of open proof goals, the collected lemmas and properties. Last but not least it offers a range of automatic decision or semi-decision procedures in special cases. Well-known examples of theorem provers are Isabelle [Pau93], Coq [Coq98], *PVS* [ORS92], and HOL [GM93].

To assure rigorous formal reasoning, we employ the theorem prover *PVS* (Prototype Verification System) developed at SRI International Computer Science Laboratory. *PVS* is written in Common Lisp and has been used for a wide range of applications; see [Rus01] for an extensive bibliography.

PVS's built-in specification language is a typed higher-order logic, extended with predicate subtypes and dependent types. Type declarations, their operations and properties are bundled together into so-called *theories* which can be organized hierarchically using the `IMPORTING` construct. Theories may contain declarations, definitions, axioms, lemmas, and theorems, and can be parameterized with type or value parameters. *PVS* has a extensive prelude with many predefined types such as natural numbers, integers, reals, sets, relations, functions, etc., and associated lemmas about their properties. Type construction mechanisms are available for building complex types, e.g., lists, function types, records, and recursively defined abstract data types. Being based on a typed logic, *PVS* automatically performs type-checking to ensure consistency of the specification and the proof-in-progress. Furthermore, the type checking mechanism generates new proof obligations, so-called *Type-Correctness Conditions* (TCCs), which are often very useful for an early detection of inconsistencies.

Besides the typed internal logic, the *PVS* environment supports the interactive verification by predefined and user-definable proof strategies. It offers facilities for proof maintenance, such as editing and rerunning (partial) proofs, easy reuse of already existing proofs, and the like. *PVS* notation will be introduced when used in the examples; for a complete description of *PVS* we refer to the *PVS* manual [OSRSC99]. In the sequel, the `typewriter` font indicates formalization in the *PVS* language.

9.2 *Verger*

In the following we apply the proof system to examples, using the *Verger* tool. As already mentioned, the tool generates the verification conditions in *PVS* syntax for an input proof outline. The tool checks also for syntactical and type correctness of the input proof outline. Furthermore, it checks if the proof outline fulfills the restrictions introduced in the previous sections, for example that actual parameters do not contain instance variables, that formal parameters are not assigned to, that the result of communication is not assigned to instance variables, etc. *Verger* generates also the weakest precondition of assignments, if required, which can be indicated by empty assertions /*{}*/. *Verger* allows also partial annotation; missing assertions are by definition true.

Since the state changes caused by a computation step are represented in the proof system by substitutions, we only need to encode the semantics of the assertion language in *PVS* (cf. Figure 1.2 on page 7). The more semantic approaches based on the global store model [AL97, JKW03, vON02] require an explicit encoding of the programming language semantics.

Our experience shows, that most of the work must be put into the definition of proof outlines; verification conditions which does not contain quantification, could be usually shown automatically using *PVS*'s `grind` strategy.

While the proof system is introduced for an abstract language, the tool supports programs in *Java*- and annotations in an adaptation of the *JML*-syntax. There are some minor syntactical differences: For example, *Java* uses = for assignments, and $==$ for equality. Conditional expressions $\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}$ of the abstract syntax are denoted in *Java* by $(b?e_1 : e_2)$. Quantification in *JML*-syntax ($\backslash$*forall* $t\,z;\ p_1;\ p_2$) expresses that all values z of type t with the property p_1 satisfy p_2, where ($\backslash$*exists* $t\,z;\ p_1;\ p_2$) expresses that there is a value z of type t with the property p_1, which satisfies p_2. The logical operators && and || of *JML* (and *Java*) correspond to $\wedge$ and $\vee$.

Augmentation and annotation are represented by special comments: Augmentations $\langle stm\rangle$ can be inserted in the *Java* program as special comments of the form $/^*\langle stm\rangle^*/$. Augmentations $\langle stm\rangle^{new}$, $\langle stm\rangle^{ass}$, etc. are represented by $/^*{new}\langle stm\rangle^*/$, $/^*{ass}\langle stm\rangle^*/$, etc. Multiple assignments $\langle \vec{y} := \vec{e}\rangle$ are syntactically represented by $/^*\langle y_1 = e_1; \ldots, y_n = e_n;\rangle^*/$.

The syntax of annotation is similar: we use $/^*\{p\}^*/$ instead of the notation $\{p\}$ of the theoretical part; furthermore, we use $/^*{new}\{p\}^*/$ instead of $\{p\}^{new}$, $/^*{ass}\{p\}^*/$ instead of $\{p\}^{ass}$, etc.

9.2.1 Representation of states in *PVS*

Before dealing with verification conditions, let us have a look how objects are represented in *PVS*. Besides a theory defining objects, two additional theories are generated for each class: One defining the reference type, and one specifying the state of class instances. This way, the classes can use each other's type definition without mutual dependency.

For the class `class c {int x;...}`, *Verger* generates the following type definitions:

```
Object: THEORY
BEGIN
  null: int
  Object_type: NONEMPTY_TYPE = {p:PRED[int] | p(null)}
    CONTAINING (LAMBDA (i:int): TRUE)
  Object?: Object_type
  Object: NONEMPTY_TYPE = (Object?) CONTAINING null
  class_name: NONEMPTY_TYPE = {cn:string | cn = "c"}
    CONTAINING "c"
  class: [Object->class_name]
  Thread: NONEMPTY_TYPE = Object CONTAINING null
END Object

c_type: THEORY
BEGIN
  IMPORTING Object
  c?: [Object->bool] = LAMBDA (i:Object):
    i=null OR class(i)="c"
  c: NONEMPTY_TYPE = (c?) CONTAINING null
  c_nn: TYPE = {i:c | i/=null}
END c_type

c: THEORY
BEGIN
  IMPORTING c_type
  x : [c_nn -> int] ...
END c
```

General specifications of classes and objects are grouped into the theory `Object`. Object identifiers are represented by the integers. Constants of a given type can be declared by `<name>:<type>`; for example `null:int` specifies the null reference as an object identifier constant. Object identifier sets, specified by predicates over integers and containing the null reference, build the domain for the type `Object_type`, where `Object?` is a value from this domain which corresponds to $Val_{null}^{\mathsf{Object}}(\sigma)$. `Object` is a type with domain `Object?`. The type `class_name` specifies the names of the classes in a program, and `class` is a function assigning class names, i.e., types, to the existing objects. The type `Threads` is an abbreviation for the type `Object`.

The theory `c_type` specifies the corresponding class type. The domain `c?` of the type `c` implements the set $Val_{null}^{c}(\sigma)$ of the semantics and consists of all existing objects of type `c` and the null reference; `c_nn` excludes null.

Finally, the theory `c` specifies states of `c`-instances: The values of the integer instance variable `x` of existing instances of the class are specified by the function `x` assigning to each existing object of type `c_nn` an integer `x`-value. Note that the representation differs from that of global states: A global state assigns to each existing object an instance state, which again assigns values to the instance variables. In the *PVS* representation, for each instance variable `x` of a class `c`, we assign to each object `o` of type `c_nn` a value `c.x(o)` from the corresponding domain, where the *PVS* expression `c.x(o)` denotes the application of the function `x` in theory `c` to `o`.

The instance state definitions are used in global conditions only. Local conditions define the instance variables of the given object locally in the theories containing the verification conditions. Also local variables are represented this way.

9.2.2 Built-in augmentation

The augmentation with the built-in auxiliary variables is automatically included and is not visible to the user, but their values may be used in the user-defined augmentation and annotation.

Tuples and their types, i.e., product types, have the notation $(\mathtt{e}_1,\ldots,\mathtt{e}_n)$ and $[\mathtt{t}_1,\ldots,\mathtt{t}_n]$ in *PVS*, where the `i`th element of a tuple `s` is specified by `proj_i(s)`; however in *Java* code this notation could lead to syntactical conflicts. Thus in proof outlines we use the notation $(:\mathtt{e}_1,\ldots,\mathtt{e}_n:)$ for tuples and $[:\mathtt{t}_1,\ldots,\mathtt{t}_n:]$ for product types; `proj(s,i)` is the projection on the ith component.

For the update of lists, which are represented in *PVS* by finite sequences `finseq[t]` of values of type `t`, we need the following functions, whose *PVS* definition is automatically generated: Given a sequence `s` and an element `e`, the expression `index(s,e)` retrieves the index of an occurrence of `e` in `s`, if any, and gives `-1` otherwise. The function `choose` assigns to each non-empty sequence a non-negative integer smaller than the length of the sequence; for the empty sequence its value is `-1`. The function `get` applied to a sequence `s` of type $\mathtt{finseq[[t_1,t_2]]}$ and an element `e` of type $\mathtt{t}_1$ gives the index of an element of `s` with first component `e`, if any, and `-1` otherwise. The expression `remove(s,i)` gives `s` without its `i`th element if $0 \leq i < |s|$, and returns `s` otherwise. The predicate $e \in s$ is syntactically represented by `includes(s,e)`. The function append appends an element at the end of a sequence, and finally o concatenates two sequences. The above functions are deterministic.

In proof outlines, auxiliary variables $x_1,\ldots,x_n$ of type t can be defined by the augmentation syntax /*< t $x_1,\ldots,x_n$;>*/. Correspondingly to *Java*, auxiliary instance variables get declared in classes outside of method definitions, where auxiliary variable declarations inside of methods specify local variables. All instance and local variables must be defined at the beginning of classes and methods, respectively.

We illustrate the use of the specific auxiliary variables by the following *Java* class definition:

```
public class Annotation extends Thread{
    void m1(){ this.start(); }
    synchronized void m2(){}
    public void run(){ this.m2(); }
}
```

Verger generates the following built-in augmentation (which is not visible to the user):

```
public class Annotation extends Thread {
  /*< finseq[[:Thread,int:]] wait; >*/
  /*< finseq[[:Thread,int:]] notified; >*/
  /*< boolean started; >*/
  /*< int counter; >*/
  /*< [:Thread,int:] lock; >*/

void m1(Thread thread, [:Object,int,Thread:] caller) {
  /*< int conf; >*/
  /*?call<conf=counter; counter=counter+1;>*/
  this.start(this, (:this,conf,thread:));
  return;
}

synchronized void m2(Thread thread, [:Object,int,Thread:] caller) {
  /*< int conf; >*/
  /*?call<conf=counter; counter=counter+1; lock=(:thread,proj(lock
      ,2)+1:);>*/
  return;
  /*!ret<lock=(:proj(lock,2) == 1 ? null : proj(lock,1),proj(lock,2)
      -1:);>*/
}

public void run(Thread thread, [:Object,int,Thread:] caller) {
  /*< int conf; >*/
  /*?call<conf=counter; counter=counter+1; started=true;>*/
  this.m2(thread, (:this,conf,thread:));
  return;
} }
```

As described in the previous chapters, the class gets extended with the built-in auxiliary instance variables wait, notified, started, counter, and lock. Furthermore, each method header includes the additional auxiliary formal parameters thread and caller. Finally, each method declares an auxiliary local variable conf.

Each method invocation reserves a fresh identity for the callee local configuration by increasing the value of counter, after its old value, the callee identity, is stored in the callee's local variable conf. Remember that the lock variable stores the identity of the thread owning the lock, and the number of synchronized method executions within the given object in the stack of the lock owner. The value `(null,0)` corresponds to a free lock. The lock value gets increased upon synchronized method invocation by the observation `lock=(:thread,proj(lock,2)+1:)` of the callee, and decreased by returning from such a method. If the lock gets free after returning is computed runtime by a conditional expression: The lock is given free only if returning terminates the last synchronized method execution in the given object by the lock owner, i.e., if `proj(lock,2)==1`. Finally, starting a new thread sets `started` of the given object to `true`.

The class is further extended with the specification of the monitor methods:

```
public void wait(Thread thread, [:Object,int,Thread:] caller) {
  /*< int conf; >*/
  /*?call<conf=counter; counter=counter+1;
     wait=append(wait,lock); lock=(:null,0:);>*/
  return;
  /*!ret<lock=notified[get(notified,thread)];
     notified=remove(notified,get(notified,thread));>*/
}

public void notify(Thread thread, [:Object,int,Thread:] caller) {
  /*< int conf; >*/
  /*?call<conf=counter; counter=counter+1;>*/
  /*<wait=remove(wait,choose(wait));
     notified=(choose(wait)==-1 ? notified
                               : append(notified,wait[choose(wait)])
                                  ); >*/
  return;
}

public void notifyAll(Thread thread, [:Object,int,Thread:] caller)
     {
  /*< int conf; >*/
  /*?call<conf=counter; counter=counter+1;>*/
  /*<notified=o(notified,wait); wait=empty_seq();>*/
  return;
}
```

The statements of the monitor methods, generated by *Verger*, do not use the auxiliary statements !signal, !signal_all, and ?signal of the semantics. Instead we implement the `wait` and `notify` methods by means of auxiliary instance variables `wait` and `notified` which represent the corresponding sets of the semantics. In the augmented `wait` method both the waiting and the notified status of the executing thread are represented by a single control point. The two statuses can be distinguished by the values of the `wait` and `notified` variables.

Invoking the `wait` method gives the lock free and stores the old lock value in the wait set, which is restored if the `wait` method terminates. Remember that returning from the `wait` method is possible only if the executing thread is already notified, and if additionally the lock of the object is free. Notification using the `notify` method moves an element from the wait into the notified set. Notifying all waiting threads in the `notifyAll` method moves all elements from the wait set into the notified set.

Though we listed the specification of the monitor methods above to demonstrate the usage of the built-in auxiliary variables, these methods are not included syntactically in *Java* class definitions. The user may additionally augment and annotate the monitor methods by special comments (see the proof outline below).

9.2.3 Proof outline

To demonstrate the proof system, we will use the following class, which implements a simple account, offering interfaces for deposit and withdraw (see also Example 4.3.6). To assure that the balance `x` remains non-negative, the withdraw method is synchronized; implicitly, the balance does not get decreased between the evaluation of `x>=i` in the `withdraw` method and the withdrawal. The annotation expresses that for each class instance, under the assumption, that the methods `deposit` and `withdraw` are called with positive parameters only, the balance `x` has always a non-negative value, as stated in the class invariant, which is defined as a local assertion $/^*\{I\}^*/$ inside of the class but outside of method definitions. Functions, like *owns* and *free_for* in the example below, can be defined as special annotations outside of class definitions: $/^*\{t\ f(t_1\ x_1,\dots,t_n\ x_n)\ =\ e\}^*/$ defines a function f of type $(t_1,\dots,t_n)\to t$ with specification $f(x_1,\dots,x_n)=e$.

The monitor methods are not included syntactically in *Java* class definitions. However, they can be augmented and annotated using special comments. For example, augmentation and annotation for the `wait` method can be inserted between the comments `/*[wait]*/` and `/*[]*/`. In the example below the annotation of the `wait` method expresses that it is not called, and thus the assertions of the program need not be invariant under its built-in augmentation. Note that the built-in augmentation is inserted by the tool, i.e., it is not defined in the input proof outlines, and is not visible to the user.

```
//function definitions
/*{ boolean owns(Thread thread, [:Thread,int:] lock) =
      thread!=null && thread==proj(lock,1) }*/
/*{ boolean free_for(Thread thread, [:Thread,int:] lock) =
      thread!=null && (thread==proj(lock,1) || proj(lock,1)==null ) }*/

public class Account{
    private int x;

    /*{ x>=0 }*/ //class invariant

    //annotation of the wait method
    /*[ wait ]*/ /*?call{ false }*/ /*{ false }*/
              /*< return; >*/ /*!ret{ false }*/ /*[]*/

    private void change_balance(int i){
        /*{ i>0 || (x+i>=0 && owns(thread,lock)) }*/
        x = x+i;
        /*{ i>0 || owns(thread,lock) }*/
    }

    public void deposit(int i){
        /*{i>0}*/
        change_balance(i);
    }

    public synchronized void withdraw(int i){
        /*?call{ free_for(thread,lock) }*/
        /*{ i>0 && owns(thread,lock) }*/
        if (x>=i) {
```

```
                /*{ x>=i && i>0 && owns(thread,lock) }*/
                change_balance(-i);
                /*wait{ i>0 }*/
                /*{ owns(thread,lock) }*/
            } /*{ owns(thread,lock) }*/
            return;
            /*!ret{ owns(thread,lock) }*/
        }
}
```

For the above proof outline 26 verification conditions are generated (4 local correctness conditions, 19 interference freedom conditions, and 3 cooperation test conditions; see the following sections). All conditions have been proven automatically by *PVS*, using the `grind` strategy.

9.2.4 Initial correctness conditions

Since the above proof outline does not specify the main class of a program, we use another proof outline to demonstrate initial correctness (see also Example 2.4.8). The following proof outline consists of the specification of its main class only. Note that the static `main` method just creates an instance of the main class, starts its thread, and terminates[1]. Thus we can assume in the proof system, that the initial configuration of a proof outline contains a single instance of the main class, being in its initial state, and a single thread, executing the `run` method of the initial object. The global invariant is specified as a global assertion /*{GI}*/ outside of class definitions.

```
//global invariant
/*{(\exists Initial z1; z1!=null; (\forall Initial z2; z2!=null;
    z1==z2))}*/

public class Initial extends Thread{
  int x;

  //class invariant
  /*{ started }*/

  public static void main(String[] args){
    Initial obj;
    obj = new Initial();
    obj.start();
  }

  public void run(){
    int v;
    /*< int u; >*/
    /*?call{ u==0 && v==0 && x==0 }*/ //precondition of observation
    /*?call< u = 1; >*/             //observation of call
    /*{ u==1 && v==0 && x==0 }*/  //postcondition of observation
  }
}
```

Initial correctness requires satisfaction of the precondition p_2 of the observation at the beginning of the `run` method after initializing the values of the local and instance variables, and after initializing the formal parameters. Furthermore, the global invariant GI, the postcondition p_3 of the observation, and the class

[1]This restriction is checked syntactically by *Verger*.

invariant I of the initial object should hold after observation. Using the syntax of the previous chapters, we have to show the satisfaction of

$$\begin{aligned}\models_{\mathcal{G}} \quad & \mathsf{InitState}(z) \wedge (\forall z'.\ z' = \mathsf{null} \vee z = z') \rightarrow \\ & \qquad P_2(z)[z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][0, 0/v, u] \wedge \\ & \qquad (GI \wedge P_3(z) \wedge I(z))[1/u][z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][0, 0/v, u]\end{aligned}$$

Verger composes the above implication, carries out the substitutions, and generates the resulting condition in *PVS* syntax. For example, the assertion p_2 is $u = 0 \wedge v = 0 \wedge x = 0$. Expressing p_2 is the global language gives $P_2(z)$ defined by $u = 0 \wedge v = 0 \wedge z.x = 0$. Carrying out the substitution $(u = 0 \wedge v = 0 \wedge z.x = 0)[z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][0, 0/v, u]$ yields $0 = 0 \wedge 0 = 0 \wedge z.x = 0$, which is expressed in *PVS* syntax by `0=0 AND 0=0 AND Initial.x(z)=0`.

Verger generates the following initial condition:

```
FORALL (z:Initial) :
  (Initial.init(z) AND
  (FORALL (obj:Object) : (obj=null OR z=obj)))
IMPLIES
%precondition of observation:
  ((0=0 AND 0=0 AND Initial.x(z)=0) AND
%global invariant:
  (EXISTS (z1:Initial) : z1/=null AND
  FORALL (z2:Initial) : (z2/=null IMPLIES z1=z2)) AND
%postcondition of observation:
  (1=1 AND 0=0 AND Initial.x(z)=0) AND
%class invariant:
  true)
```

where the `init` function in theory `Initial` is defined by

```
init(o:Initial): bool = (o/=null AND Initial.x(o)=0 AND Initial.
    started(o)=false AND ...
```

9.2.5 Local correctness conditions

For the account proof outline of Section 9.2.3 *Verger* generates 4 local correctness conditions. The first one for the assignment in line 18 expresses that the class invariant together with the precondition of the assignment imply the assignment's postcondition:

```
((i>0 OR (x+i>=0 AND owns(thread,lock))) AND x>=0)
IMPLIES (i>0 OR owns(thread,lock))
```

The second one shows, that the precondition of the if-statement in line 29 in `withdraw`, the class invariant, and the boolean condition of the if-statement together imply the assertion of line 31:

```
((i>0 AND owns(thread,lock)) AND x>=0 AND x>=i)
IMPLIES (x>=i AND i>0 AND owns(thread,lock))
```

The remaining two conditions are generated for the postcondition of the if-statement. Remember that for local verification conditions, the instance and local variables are defined locally in the theories containing the lemmas.

9.2.6 Interference freedom conditions

For interference freedom, the tool implements renaming by extending the name of each local variable of the local configuration executing the assignment with _1, where the names of local variables in the assertion get extended with _2; the names of instance variables get the extension `_inst`. *Verger* does not generate conditions for trivial cases, for example if the assertion is true by definition, or if the substitution does not change the assertion.

Satisfaction of the class invariant of the example proof outline is assured by the condition

```
%precondition assignment
  ((i_1>0 OR (x_inst+i_1>=0 AND owns(thread_1,lock_inst))) AND
%class invariant
  x_inst >= 0)
IMPLIES
%class invariant after execution
  (x_inst+i_1>=0)
```

generated for the only assignment at 18, which changes the balance `x`. That the assertion at 31 is invariant under the same assignment, is assured by the condition

```
%preconditions assignment
  ((i_1>0 OR (x_inst+i_1>=0 AND owns(thread_1,lock_inst))) AND
%assertion
  x_inst>=i_2 AND i_2>0 AND owns(thread_2,lock_inst) AND
%class invariant
  x_inst >= 0 AND
%interleavable
  (thread_1=thread_2 IMPLIES false) AND (thread_1/=thread_2 IMPLIES
      true))
IMPLIES
%assertion after execution
  (x_inst+i_1>=i_2 AND i_2>0 AND owns(thread_2,lock_inst))
```

If `i_1>0`, then `x_inst>=i_2` implies `x_inst+i_1>=i_2`, and the condition is satisfied, which corresponds to the concurrent execution of the methods `withdraw` and `deposit`. Otherwise, `thread_1/=thread_2`, `owns(thread_1,lock_inst)`, and `owns(thread_2,lock_inst)` lead to a contradiction. This case corresponds to the concurrent execution of `withdraw`, which is not possible. There is a similar condition for the case that two threads are concurrently executing the `change_balance` method, showing that the assertion at 17 is invariant under the execution of the assignment at 18.

The remaining conditions are all generated for invariance under changing the lock value. There are altogether 6 assertions at control points, which have to be shown invariant under entering and exiting the `wait` method. As the `wait` method, however, is not invoked, as expressed by its annotation, the left-hand side of the generated implications is false.

The only remaining assignments changing the lock value are the observations at the beginning and at the end of the synchronized `withdraw` method. Assertions in that method which are not at a control point waiting for return, does not have to be invariant under the execution of withdraw. Thus only the assertions at 17 and at 19 in `change_balance` have to be shown invariant, which yields 4 conditions. For invariance of the assertion at 17 under entering the `withdraw` method we get:

```
%precondition assignment
  (free_for(thread_1,lock_inst) AND
%assertion
  (i_2>0 OR (x_inst+i_2>=0 AND owns(thread_2,lock_inst))) AND
%class invariant
  x_inst>=0 AND
%interleavable
  (thread_1=thread_2 IMPLIES false) AND (thread_1/=thread_2 IMPLIES
      true))
IMPLIES
%assertion after execution
  (i_2>0  OR (x_inst+i_2>=0 AND owns(thread_2,(thread_1,(PROJ_2(
      lock_inst)+1)))))
```

Note that `free_for(thread_1,lock_inst)`, `owns(thread_2,lock_inst)`, and `thread_1/=thread_2` together lead to a contradiction: If a thread executing the private `change_balance` method owns the lock, then no other thread can enter the synchronized `withdraw` method. The remaining three conditions are analogous.

9.2.7 Cooperation test for communication

Next we apply the cooperation test to the account example. Renaming is implemented by extending the name of each local variable of the caller with `_1`, where local variables of the callee get extended with `_2`. The *PVS* expression `c.x(z)` represents the qualified reference $z.x$ for z of type c.

Three cooperation conditions are generated: one for the method call in line 24, one for the call at 32, and one for the corresponding return from the second call. Note that we do not have any conditions for returning from the first call at 24, because all postconditions are by definition true. The first condition

```
FORALL (caller:Account) : caller/=null IMPLIES
FORALL (callee:Account) : callee/=null IMPLIES
%precondition caller
  ((i_1>0 AND
%class invariant caller and callee + caller-callee relationship
  Account.x(caller)>=0 AND Account.x(callee)>=0 AND caller=callee)
IMPLIES
%postcondition callee
  (i_1>0 OR (Account.x(callee)+i_1>=0 AND owns(thread_1,Account.
      lock(callee)))))
```

states that the class invariants and the preconditions of caller and callee imply the postcondition of the callee. Note that the global invariant, the postcondition of the caller, and the assertions at the auxiliary points are by definition true. The caller-callee relationship of the partners is assured by requiring `caller=callee`, since it is a self-call. The condition for the second call is similar. The condition for return assures the caller-callee relationship of the partners by additionally requiring, that the formal parameters equal the actual ones. Applied to the built-in auxiliary parameter `thread`, this requirement implies for example that caller and callee are the same thread, i.e., `thread_1=thread_2`, which we need to show that the caller owns the lock after communication:

```
FORALL (caller:Account) : caller/=null IMPLIES
FORALL (callee:Account) : callee/=null IMPLIES
%precondition caller
  ((i_1>0 AND
%class invariant caller
  Account.x(caller)>=0 AND
%precondition callee
  (i_2>0 OR owns(thread_2,Account.lock(callee))) AND
%class invariant callee
  Account.x(callee)>=0 AND
%caller-callee relationship
  caller=callee AND i_2=(-i_1) AND thread_2=thread_1 AND caller_2=(
      caller,conf_1,thread_1))
IMPLIES
%postcondition caller
  owns(thread_1,Account.lock(caller)))
```

9.2.8 Cooperation test for object creation

Finally, to demonstrate the cooperation test for object creation, the proof outline below specifies two classes, called `Creator` and `Created`. Instances of the `Creator` class offer the method `create()` which creates an instance of the `Created` class and gives it back as a return value. The global invariant states that there exists at most one instance of the `Creator` class, and that its auxiliary instance variable `nr` stores the number of the existing `Created` instances.

```
//function definition
/*{ boolean disjunct(finseq[Created] z) =
     (\forall int i; 0<=i && i<length(z); z[i]!=null &&
     (\forall int j; 0<=j && j<length(z) && i!=j ; z[i]!=z[j]))}
          */

//global invariant
/*{ (\forall Creator o; o!=null;
    (\forall Creator o2; o2!=null; o2==o) &&
    (\forall finseq[Created] z; disjunct(z) && (\forall Created
        z2; z2!=null; includes(z,z2)); o.nr == length(z)))
    }*/

class Creator {
    /*< int nr; >*/

    public Created create(){
        Created u;
        u = new Created();
        /*new< nr = nr + 1; >*/
        return u;
    }
}

class Created {}
```

We apply the proof system to these two classes. Of course, the global invariant describes a program, which contains these classes, only then correctly, if the context of these classes also preserve it. Thus we verify these classes to be correct under the assumption that the remaining verification conditions hold for the environment. *Verger* generates the following cooperation test condition for the object creation statement, where the domain of the type `Object_old` in theory `Object` is the value of the logical variable z' in the cooperation test. Correspondingly for the type of the newly created instance, the type `Created_old` covers all existing instances of the `Created` class but the new object.

```
Object: THEORY
BEGIN ...
    new_Object: Object
    Object_old: NONEMPTY_TYPE = {o:Object | o=null OR o /=
        new_Object} CONTAINING null
END Object

Created_type: THEORY
BEGIN ...
    Created_old: NONEMPTY_TYPE = {o:Created | o=null OR o /=
        new_Object} CONTAINING null
END Created_type

Created: THEORY
BEGIN ...
    init(o:Created): bool = (o=new_Object AND o/=null AND ... AND
        Created.lock(o)=(null,0))
END Created
...
global_cond_0 : THEORY
BEGIN ...
condition : LEMMA
%z/=null /\
  FORALL (creator:Creator)  : creator/=null IMPLIES
%z/=u /\ Fresh(z',u)
  ((creator/=u AND Created.init(u) AND
%GI restricted to z'
  (FORALL (o:Creator)  : o/=null IMPLIES
  ((FORALL (o2:Creator)  : o2/=null IMPLIES o2=o) AND
  (FORALL (z:finseq[Created_old])  : ((disjunct(z) AND
  (FORALL (z2:Created_old)  : (z2/=null IMPLIES includes(z,z2))))
      IMPLIES
  (Creator.nr(o)=length(z)))))))
IMPLIES
%GI after execution
  (FORALL (o:Creator)  : (o/=null IMPLIES
  ((FORALL (o2:Creator)  : (o2/=null IMPLIES o2=o)) AND
  (FORALL (z:finseq[Created])  : ((disjunct(z) AND
  (FORALL (z2:Created)  : (z2/=null IMPLIES includes(z,z2))))
      IMPLIES
  (IF (o=creator) THEN (Creator.nr(creator)+1) ELSE Creator.nr(o)
      ENDIF = length(z))))))))

END global_cond_0
```

In the antecedent of the cooperation test, quantifications in the global invariant $GI \downarrow z'$ are restricted to objects existing already before the creation. You can see in the above example, how this restriction is applied: the quantification `FORALL (z2:Created):p` got replaced by `FORALL (z2:Created_old):p`. Note that this replacement applies also to composed types: the quantification `FORALL (z:finseq[Created]):p` is replaced by `FORALL (z:finseq[Created_old]):p`.

9.2.9 Properties of the `wait` method

The following example illustrates properties of the `wait` method.[2] The `wait` method can be called only by a thread owning the lock of the callee object, as expressed by the assertion in line 24. After invoking `wait`, the thread gives the

[2] We currently do not handle exceptions in $Java_{synch}$ and its proof theory. To call the `wait` method, however, we must syntactically catch `InterruptedExceptions`. But, since we do not support the interrupt method, it cannot be thrown.

lock free, as formalized in the assertion in line 26. When returning, it becomes the lock owner again, as stated by the predicate in line 27. The tool generates 16 verification conditions for the proof outline below (14 interference freedom and 2 cooperation test conditions). All conditions are proven in *PVS*.

```
1  /*{ boolean owns(Thread thread, [:Thread,int:] lock) =
2         thread!=null && proj(lock,1)==thread }*/
3  /*{ boolean not_owns(Thread thread, [:Thread,int:] lock) =
4         thread!=null && proj(lock,1)!=thread }*/
5  /*{ boolean free_for(Thread thread, [:Thread,int:] lock) =
6        thread!=null && (thread==proj(lock,1) || lock==(:null,0:))
              }*/
7  /*{ boolean disjunct(finseq[[:Thread,int:]] x) =
8        (\forall int i,j; 0<=i && 0<=j && i<length(x)&& j<length(x)
              && i!=j; proj(x[i],1)!=proj(x[j],1)) }*/

10 public class Monitor{
11      /*< finseq[[:Thread,int:]] x;>*/

13      /*{ disjunct(x) }*/ //class invariant

15      /*[wait]*/  /*?call{ owns(thread,lock) }*/
16        /*{ not_owns(thread,lock) && proj(caller,1)==this &&
              includes(x,(:thread,proj(caller,2):)) }*/
17        /*<return;>*/
18        /*!ret{ lock==(:null,0:) && proj(caller,1)==this &&
              includes(x,(:thread,proj(caller,2):)) && get(notified,
              thread)!=-1 }*/
19      /*[]*/

21      public synchronized void m(){
22          /*?call{ free_for(thread,lock) && (\forall int i;true;!
                includes(x,(:thread,i:))) }*/
23          /*?call< x=append(x,(:thread,counter:)); >*/
24          /*{ owns(thread,lock) && includes(x,(:thread,conf:)) }*/
25          try{ this.wait();} catch (InterruptedException e){}
26          /*wait{ not_owns(thread,lock) && includes(x,(:thread,
                conf:)) }*/
27          /*{ owns(thread,lock) && includes(x,(:thread,conf:)) }*/
28          return;
29          /*!ret{ owns(thread,lock) && includes(x,(:thread,conf:))
                }*/
30          /*!ret< x=remove(x,index(x,(:thread,conf:))); >*/
31      }
32 }
```

We use the auxiliary instance variable `x` to store for each local configuration executing `m` the thread and local configuration identities. We use this information to identify local configurations in caller-callee relationship: We can exclude from the interference freedom test for example the invariance of the assertion at 26 under the built-in return-observation of its callee, setting the lock owner to the identity of the executing thread. Clearly, the assertion at 26 would not be invariant under the return-observation of its callee; caller and callee execute a common step, and the control point of the caller moves from 26 to 27. We get the following interference freedom condition for this setup, where the case `thread_1=thread_2` leads to a contradiction:

```
%precondition assignment
  (lock_inst=(null,0) AND PROJ_1(caller_1)=this AND includes(x_inst
      ,(thread_1,PROJ_2(caller_1))) AND get(notified_inst,thread_1)
      /=(-1) AND
%assertion
```

```
  not_owns(thread_2,lock_inst) AND includes(x_inst,(thread_2,conf_2
      )) AND
%class invariant
  disjunct(x_inst) AND
%interleavable
  (thread_1=thread_2 IMPLIES (conf_1/=conf_2 AND (this/=PROJ_1(
      caller_1) OR conf_2/=PROJ_2(caller_1)))) AND (thread_1/=
      thread_2 IMPLIES true))
IMPLIES
%assertion after execution
  (not_owns(thread_2,seq(notified_inst)(get(notified_inst,thread_1)
      )) AND includes(x_inst,(thread_2,conf_2)))
```

For the cooperation test, we handle only the condition for the invocation of the `wait` method at 25; we have a similar condition for the corresponding return case.

```
FORALL (caller:Monitor) : caller/=null IMPLIES
FORALL (callee:Monitor) : callee/=null IMPLIES
%precondition caller
  (owns(thread_1,Monitor.lock(caller)) AND includes(Monitor.x(
      caller),(thread_1,conf_1)) AND
%class invariant caller
  disjunct(Monitor.x(caller)) AND
%class invariant callee
  disjunct(Monitor.x(callee)) AND
%caller-callee relationship
  caller=callee AND PROJ_1(Monitor.lock(callee))=thread_1)
IMPLIES
%precondition callee observation
  (owns(thread_1,Monitor.lock(callee)) AND
%postcondition caller
  not_owns(thread_1,IF caller=callee THEN (null,0) ELSE Monitor.
      lock(caller) ENDIF) AND includes(Monitor.x(caller),(thread_1,
      conf_1)) AND
%postcondition callee
  not_owns(thread_1,(null,0)) AND PROJ_1((caller,conf_1,thread_1))=
      callee AND includes(Monitor.x(callee),(thread_1,PROJ_2((
      caller,conf_1,thread_1)))))
```

After renaming the local variables, the precondition of the method invocation directly implies `owns(thread_1,Monitor.lock(caller))`, and thus the precondition of the callee observation after substituting the actual parameter `thread_1` for the formal one `thread_2`. This implication means, that if the caller thread owns the lock, then, since the caller and the callee threads are the same, the callee thread owns the lock, too. The built-in augmentation at the beginning of the `wait` method releases the lock of the callee object. Since caller and callee object are the same, after substituting for the built-in augmentation and for communication, the caller precondition also directly implies the postconditions of both the caller and the callee.

9.3 Conclusions and related work

In this chapter we described the theorem prover *PVS* and the *Verger* tool, which generates for an input proof outline the verification conditions in the syntax of *PVS*. The use of the tool is demonstrated on some examples.

Our experience shows that most of the human effort must be put into the specification of proof outlines, i.e., into the augmentation and the annotation.

The verification conditions, which are generated as separate logical implications, were verified mostly automatically using the `grind` strategy of *PVS*. Only some of those conditions which contained quantification needed human interaction in the *PVS* verification process.

We did not carry out any larger case studies yet. However, we expect the above observations to hold also for larger case studies. Though for larger programs more verification conditions are generated, their proofs are independent of each other. In other words, the program size influences the number but not the complexity of single conditions.

As to further development of the tool, we plan to optimize the *PVS* type and state representations, and to work out further *PVS* strategies to increase the degree of automation. It would also be interesting to restrict the logic to a decidable subset, for which a fully automatic verification is possible within the theorem prover.

Further effort will be put into the automatic generation of assertions by means of weakest preconditions. Runtime checks of the annotations could detect non-invariant assertions at an early stage of the verification process. Cheon et al. present such an approach to runtime assertion checking of *JML* assertions in [Che03, LCC$^+$03].

The Jass tool (*Java* with assertions) [BFMW01], developed by Bartetzko et al., is a Design by Contract extension of *Java*. The tool allows runtime checks of the assertions of annotated *Java* programs. Brörkens and Möller deal with runtime checking at the bytecode level [Möl02, BM02]. The underlying framework `jasda` allows one to test the dynamic behavior of multiple *Java* virtual machines by monitoring whether the trace of all relevant events is a member of the trace semantics of a given CSP process or not.

As already mentioned, the verification conditions of our proof system are logical implications. Furthermore, we generate those verification conditions automatically using the *Verger* tool. That means, we only have to encode the semantics of assertions in the theorem prover.

In the previous Sections 2.5, 3.4, and 4.4 we discussed also more semantically-oriented approaches which define the syntax, the semantics, and the proof system for a programming language within a theorem prover. The idea of such representations goes back at least to Gordon [Gor89], who developed a Hoare logic for a simple imperative language. Using a theorem prover, the Hoare rules are mechanically derived from the programming language semantics. These rules form the basis for a simple program verifier.

Theorem prover are not only used to show correctness of *Java* source code. For example, Basin et al. [BFPV99, BFGP02] present a model checking algorithm and its implementation in *Isabelle/HOL* to check type correctness of *Java* bytecode. They use *Isabelle/HOL* to formalize and prove correctness of their approach [BFG02].

The Compaq Extended Static Checker for *Java* (ESC/Java) [ESC00] is a programming tool for finding errors in *Java* programs. ESC/Java detects, at compile time, common programming errors that ordinarily are not detected until run time, and sometimes not even then; for example, null dereference errors, array bounds errors, type cast errors, and race conditions. Detlefs et al. describe and motivate extended static checking in [DLNS98]. A verification condition generator produces logical formulas assuring that a program is free of a particular class of errors. A theorem prover is used to prove the conditions; the checker has been implemented for Modula-3. In [LSS99], Leino et al. use an intermediate guarded-command language for verification condition generation.

Chapter 10

Concluding remarks

In this thesis we presented a tool-supported assertional proof method for a *Java* sublanguage covering multithreading and *Java*'s monitor discipline. We introduced the language and the proof system incrementally in three steps: We started with a *sequential* *Java* sublanguage and its proof system. In the next step we included dynamic thread creation, resulting in a *multithreaded* sublanguage. Finally, we extended the language and the proof system to cover *monitor synchronization*. We gave proofs of soundness and relative completeness. The proof system also allows to prove deadlock freedom.

The development of the proof system was an interesting and challenging task. During this process we changed the definitions and formalizations over and over again, until we reached the current version, which clearly mirrors the semantics of the language.

We have illustrated the use of our assertional proof system on a number of examples, which have been verified using the tool *Verger*. The tool takes an augmented and annotated *Java* program, a so-called proof outline, as input and generates those verification conditions which assure invariance of the annotation. We used the theorem prover *PVS* to verify these conditions.

The verification conditions are defined by standard logical formulas, where the effect of execution is captured by substitutions. This representation requires only the embedding of the assertion semantics in the theorem prover, but not of the semantics of the programming language. The simplicity of the representation increases the automation of the proofs.

The assertional logic and the proof system are modular in that they allow one to describe and analyze object-internal and object-external execution separately. This modularity makes local proofs reusable.

Concurrency in class-based object-oriented languages is not just an extension of sequentiality, but a fundamentally new concept. The way of thinking about a program, about its structure and its behavior, is qualitatively different. The state-based approach for sequential programs must be extended with an interface-based approach.

On the one hand, the complexity of a sound and relatively complete Hoare-style proof system for a programming language immediately reflects the complexity of the semantics of that language. In the case of multithreaded *Java*, it is doable to extend our proof system to, e.g., the Java Memory Model (see Section 8.1). However, the sheer size of technical machinery involved indicates that, without massive computer support, the limits of this style of proof systems have been reached.

On the other hand, the complexity of a proof system for a programming language is inversely proportional to the chance that programs written in that language are correct. That means, not only the correctness proofs of concurrent *Java* programs are complex, but it is also hard to develop correct multithreaded *Java* programs.

A natural solution to reduce the complexity of the behavior and of the verification procedure for concurrent programs could be to restrict interference between different threads, for example using synchronization. From the view point of the semantics, such a restriction would allow a better understanding of program behavior and would make it easier to write correct programs. From the verification side, it would slow down the exponential increase of the number of verification conditions for increasing program size.

Future work The preceding chapter on possible extensions shows that there are a lot of challenging and interesting research topics in the field, which need further analysis.

The incremental development illustrated how to extend the language and the proof system to deal with additional language features. As to future work, we plan to extend the programming language by further constructs, like inheritance, subtyping, and exception handling. Since these extensions naturally increase the complexity of the proof system, further development of the tool is highly important. Restricting the logic to a decidable subset would allow fully automatic proof of the verification conditions.

Computer support for the specification of proof outlines, i.e., for the augmentation and annotation, would be of great practical relevance. The specification of the annotation could use a weakest precondition calculus. However, due to concurrency, those annotations are in general not yet inductive. I.e., they must be made stronger in order to exclude from the interference freedom test those pairs of control points which are not simultaneously reachable.

We are also interested in the development of a compositional proof system, based on a compositional semantics [ÁdBdRS04a].

Bibliography

[ABB+00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In Ojeda-Aciego et al. [OAdGBP00], pages 21–36.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[ACM03] ACM. *ACM Conference on Programming Language Design and Implementation*, June 2003.

[ACR98] Isabelle Attali, Denis Caromel, and Marjorie Russo. A formal executable semantics for Java. In OOPSLA'98 [OOP98]. In *SIGPLAN Notices* 30(10).

[AdB90a] Pierre America and Frank S. de Boer. A proof system for process creation. In *IFIP TC-2 Working Conference on Programming Concepts and Methods*, pages 303–332, 1990.

[AdB90b] Pierre America and Frank S. de Boer. A sound and complete proof system for SPOOL. Technical Report 505, Philips Research Laboratories, 1990.

[AdB93] Pierre America and Frank S. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1993.

[ÁdBdRS03a] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A Hoare logic for monitors in Java. Techical report TR-ST-03-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, April 2003. Available at `http://www.informatik.uni-kiel.de/inf/deRoever/techreports/03/tr-st-03-%1.pdf`.

[ÁdBdRS03b] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof-outlines for monitors in Java. In Najm et al. [NNS03], pages 155–169. A longer version appeared as technical report TR-ST-03-1, April 2003 (`http://www.informatik.uni-kiel.de/inf/deRoever/techreports/03/tr-st-03%-1.pdf`).

[ÁdBdRS03c] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof outlines for multithreaded Java with exceptions. Technical Report 0313, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, December

2003. Available at `http://www.informatik.uni-kiel.de/reports/2003/0313.html`.

[ÁdBdRS03d] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A tool-supported assertional proof system for multithreaded Java. In Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, and Erik Poll, editors, *Proc. of the Workshop on Formal Techniques for Java-like Programs - FTfJP'2003*, 2003.

[ÁdBdRS04a] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A compositional operational semantics for Java$_{MT}$. In Derschowitz [Der04], pages 290–303. A preliminary version appeared as Technical Report TR-ST-02-2, May 2002.

[ÁdBdRS04b] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof outlines for exceptions in multithreaded Java. 2004. Submitted for publication, June 2004.

[AF99] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science State-of-the-Art-Survey*. Springer-Verlag, 1999.

[AF03] Thomas Arts and Wan Fokkink, editors. *Eighth International Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

[AFdR80] Krzysztof R. Apt, Nissim Francez, and Willem-Paul de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[AFL99] Jim Alves-Foss and Fong Shing Lam. Dynamic denotational semantics of Java. In Alves-Foss [AF99], pages 201–240.

[AH00] Mark Aagaard and John Harrison, editors. *Theorem Proving in Higher Order Logics (TPHOL 2000)*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[AJ01] Isabelle Attali and Thomas Jensen, editors. *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, 2001.

[AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Bidoit and Dauchet [BD97], pages 682–696. An extended version of this paper appeared as SRC Research Report 161 (September 1998).

[ÁMdB00] Erika Ábrahám-Mumm and Frank S. de Boer. Proof-outlines for threads in Java. In Palamidessi [Pal00], pages 229–242.

[ÁMdBdRS01] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Deductive verification for multithreaded Java (extended abstract). In *Proceedings of the "11. Kolloquium Programmiersprachen und Grundlagen der Programmierung", 2001, Rurberg*, pages 121–126, 2001.

[ÁMdBdRS02a] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A compositional operational semantics for

Java$_{MT}$. Technical Report TR-ST-02-2, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, May 2002.

[ÁMdBdRS02b] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A tool-supported proof system for monitors in Java. In Bonsangue et al. [BdBdRG03], pages 1–32.

[ÁMdBdRS02c] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In Nielsen and Engberg [NE02], pages 4–20. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.

[ÁMdBdRS02d] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept: Soundness and completeness. Technical Report TR-ST-02-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 2002.

[Ame89] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. 443, Phillips Research Laboratories, January/April 1989.

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[Apt81a] Krzysztof R. Apt. Recursive assertions and parallel programs. *Acta Informatica*, 1981.

[Apt81b] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[Apt83] Krzysztof R. Apt. Formal justification of a proof system for communicating sequential processes. *Communications of the ACM*, 30(1):197–216, January 1983.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[Ash75] Edward A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[Bal03] The project Bali. `http://isabelle.in.tum.de/Bali/`, 2003.

[BCM00] Didier Bert, Christine Choppy, and Peter Mosses, editors. *Recent Trends in Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[BD97] Michel Bidoit and Max Dauchet, editors. *Theory and Practice of Software Development, Proceedings of the 7th International Joint Conference of CAAP/FASE, TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, Lille, France, April 1997. Springer-Verlag.

[BdBdRG03] Marcello M. Bonsangue, Frank S. de Boer, Willem-Paul de Roever, and Susanne Graf, editors. *Proceedings of the First International*

Symposium on Formal Methods for Components and Objects (FMCO 2002), Leiden, volume 2852 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In Attali and Jensen [AJ01], pages 6–24.

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[BFG02] David Basin, Stefan Friedrich, and Marek Gawkowski. Verified bytecode model checkers. In Carreño et al. [CMT02], pages 47–66.

[BFGP02] David Basin, Stefan Friedrich, Marek Gawkowski, and Joachim Posegga. Bytecode model checking: An experimental analysis. In Bošnački and Leue [BL02], pages 42–59.

[BFMW01] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with assertions. In Havelund and Rosu [HR01].

[BFPV99] David Basin, Stefan Friedrich, Joachim Posegga, and Harald Vogt. Java byte code verification by model checking. In Halbwachs and Peled [HP99], pages 491–494.

[BGZ98] Lubos Brim, Jozef Gruska, and Jirí Zlatuska, editors. *International Symposium on Mathmatical Foundations of Computer Science (MFCS)*, volume 1450 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[BH03] Richard Bubel and Reiner Hähnle. Formal specification of security-critical railway software with the key system. In Arts and Fokkink [AF03]. Available at `http://johann.math.tulane.edu/~entcs/`.

[BL02] Dragan Bošnački and Stefan Leue, editors. *Model Checking Software, 9th International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2002.

[BM02] Mark Brörkens and Michael Möller. Dynamic event generation for runtime checking using the JDI. In Havelund and Rosu [HR02].

[Bör99] Egon Börger, editor. *Architecture Design and Validation Methods*. Springer-Verlag, 1999.

[BP02] Manfred Broy and Markus Pizka, editors. *Models, Algebras and Logic of Engineering Software, Summer School (Marktoberdorf, Germany, 2002)*, volume 191 of *Series F: Computer and System Sciences*. NATO Advanced Study Institute, IOS Press, 2002.

[BS89] Graham Birtwhistle and Pasupati A. Subrahmanyam, editors. *Current Trends in Hardware Verification and Automated Theorem Proving*, number 15 in Workshops in Computing. Springer-Verlag, 1989.

[BS97] Rudolf Berghammer and Friedeman Simon, editors. *Proceedings of Programming Languages and Fundamentals of Programming*. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, November 1997. Bericht Nr. 9717.

[BS98] Egon Börger and Wolfram Schulte. Defining the Java Virtual Machine as platform for provably correct Java compilation. In Brim et al. [BGZ98], pages 17–35.

[BS99a] Egon Börger and Wolfram Schulte. Modular design for the Java virtual machine architecture. In Börger [Bör99], pages 297–356.

[BS99b] Egon Börger and Wolfram Schulte. A programmer-friendly modular definition of the semantics of Java. In Alves-Foss [AF99], pages 353–404.

[BS00] Egon Börger and Wolfram Schulte. A practical method for specification and analysis of exception handling - a Java/JVM case study. *IEEE Transactions on Software Engineering*, 26(10):872–887, October 2000.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines - A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[BSBR03] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation (San Diego, California)* [ACM03].

[Cen99] Pietro Cenciarelli. Towards a modular denotational semantics of Java. In Moreira and Demeyer [MD99], page 105.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[Che03] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report TR-03-09, Department of Computer Science, Iowa State University, April 2003.

[CKRW97] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From sequential to multi-threaded Java: An event-based operational semantics. In Johnson [Joh97], pages 75–90.

[CKRW99] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [AF99], pages 157–200.

[CMT02] Victor A. Carreño, César A. Muñoz, and Sofiène Tahar, editors. *Theorem Proving in Higher Order Logics (TPHOLs'02)*, volume 2410 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2002.

[Coo78] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, February 1978.

[Coq98] The Coq project. `http://pauillac.inria.fr/coq/`, 1998.

[CR98] Eva Coscia and Gianna Reggio. A proposal for a semantics of a subset of multi-threaded good Java. In Susan Eisenbach, editor, *Proceedings of the Workshop on the formal Underpinnings of Java, Vancouver*, 1998.

[CR99] Eva Coscia and Gianna Reggio. An operational semantics for Java. Technical Report DISI-TR-99-06, DISI- UniversitÃ di Genova, 1999.

[CT00] Thomas W. Christopher and George K. Thiruvathukal. *High-Performance Java Platform Computing: Multithreaded and Networked Programming*. Prentice Hall PTR and Sun Microsystems Press, 2000.

[CW96] Mary Campione and Kathy Walrath. *The Java Tutorial*. The Java series. Addison-Wesley, 1996. Available at `http://java.sun.com/docs/books/tutorial/`.

[dB90] Frank S. de Boer. A proof system for the parallel object-oriented language POOL. In Paterson [Pat90].

[dB91a] Frank S. de Boer. A proof system for the language POOL. In de Bakker et al. [dBdRR91], pages 124–150.

[dB91b] Frank S. de Boer. *Reasoning about Dynamically Evolving Process Structures. A Proof Theory for the Parallel Object-Oriented Language POOL*. PhD thesis, Free University of Amsterdam, 1991.

[dB99] Frank S. de Boer. A WP-calculus for OO. In Thomas [Tho99], pages 135–156.

[dBdRR91] Jaco W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors. *Foundations of Object-Oriented Languages (REX Workshop)*, volume 489 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[dBdRR94] Jaco W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors. *A Decade of Concurrency 1993 (REX Workshop)*, volume 803 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[dBP02] Frank S. de Boer and Cees Pierik. Computer-aided specification and verification of annotated object-oriented programs. In Jacobs and Rensink [JR02], pages 163–177.

[dBP03] Frank S. de Boer and Cees Pierik. Towards an environment for the verification of annotated object-oriented programs. Technical report UU-CS-2003-002, Institute of Information and Computing Sciences, University of Utrecht, January 2003.

[DEK99] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[Der04] Nachum Derschowitz, editor. *Proceedings of the International Symposium on Verification (Theory and Practice), Celebrating Zohar Manna's 64th Birthday, Taormina, Sicily, June 29–July 4, 2003*, volume 2772 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[dF95] Carlos Camarao de Figueiredo. A proof system for a sequential object-oriented language. Technical Report UMCS-95-1-1, University of Manchester, 1995. The technical report corresponds the author's PhD thesis.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DIS98] Claudio Demartini, Radu Iosif, and Riccardo Sisto. Modeling and validation of Java multithreading applications using SPIN. In Najm et al. [NSH98].

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Technical Note 159, Compaq Systems Research Center, December 1998. Available at `http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/`.

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods.* Cambridge University Press, 2001.

[DS90] Edsger W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[DV00] Sophia Drossopoulou and Tatyana Valkevych. Java exceptions throw no surprises. Technical report, Dept. of Computing, Imperial College of Science, London, 2000.

[ECO00] *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000), Sophia Antipolis and Cannes*, volume 1850 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2000.

[EL02] Lars-Henrik Eriksson and Peter A. Lindsay, editors. *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002.

[ESC00] Extended static checking for Java. `http://research.compaq.com/SRC/esc/`, 2000.

[FFQ02] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In Métayer [Mét02], pages 262–277.

[FKF99] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In Alves-Foss [AF99], pages 241–269.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.

[FOO02] *Proceedings of the 9th International Workshop on Foundations of Object-Oriented Languages (FOOL'02)*, 2002.

[GdR98] David Gries and Willem-Paul de Roever, editors. *Programming Concepts and Methods (PROCOMET '98).* International Federation for Information Processing (IFIP), Chapman & Hall, 1998.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[GJSB00] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, Second edition, 2000.

[GKOT00] Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors. *Abstract State Machines: Theory and Applications*, volume 1912 of *lncs*. Springer-Verlag, 2000.

[GM93] Michael J. C. Gordon and Thomas F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[Gor89] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. In Birtwhistle and Subrahmanyam [BS89], pages 387–439.

[GPS02] Alex Gontmakher, Sergey Polyakov, and Assaf Schuster. Complexity of verifying Java shared memory executions. *Parallel Processing Letters*, 2002.

[Gra97] Mark Grand. *Java Language Reference*. O'Reilly, Second edition, 1997.

[GS00] Alex Gontmakher and Assaf Schuster. Java consistency: non-operational characterizations for Java memory behavior. *ACM Transactions On Computer Systems (TOCS)*, 18(4):333–386, 2000.

[GSW00a] Yuri Gurevich, Wolfram Schulte, and Charles Wallace. Investigating Java concurrency using Abstract State Machines. In Gurevich et al. [GKOT00], pages 151–176.

[GSW00b] Yuri Gurevich, Wolfram Schulte, and Charles Wallace. Investigating Java concurrency using Abstract State Machines. Technical Report 2000-04, University of Delaware, 2000.

[GZ98] Sabine Glesner and Wolf Zimmermann. Using many-sorted natural semantics to specify and generate semantic analysis. In SI2000 [SI298], pages 249–262.

[HJ00] Marieke Huisman and Bart Jacobs. Inheritance in higher order logic: Modeling and reasoning. In Aagaard and Harrison [AH00], pages 301–319.

[HJvdB01] Marieke Huisman, Bart Jacobs, and Joachim van den Berg. A case study in class library verification: Java's vector class. *Software Tools for Technology Transfer*, 3(3):332–352, 2001.

[HM01] Pieter H. Hartel and Luc Moreau. Formalizing the safety of Java, the Java virtual machine, and Java Card. *ACM Computing Surveys*, 33(4):517–558, 2001.

[Hoa69] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa74] Charles A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoa78] Charles A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol00] Allen Holub. *Taming Java Threads*. Apress, 2000.

[HP99] Nicolas Halbwachs and Doron Peled, editors. *CAV '99: Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[HR01] Klaus Havelund and Grigore Rosu, editors. *RV'2001 Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[HR02] Klaus Havelund and Grigore Rosu, editors. *RV'2002 Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[Hui01] Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[Hus01] Heinrich Hussmann, editor. *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Hyd01] Paul Hyde. *Java Thread Programming*. SAMS Publishing, 2001.

[IDS99] Radu Iosif, Claudio Demartini, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, July 1999.

[IP00] Atsushi Igarashi and Benjamin Pierce. On inner classes. In ECOOP2000 [ECO00], pages 129–154.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In OOPSLA'99 [OOP99], pages 132–146. In *SIGPLAN Notices*.

[Jac01] Bart Jacobs. A formalisation of Java's exception mechanism. In Sands [San01], pages 284–301.

[JKW03] Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In Bonsangue et al. [BdBdRG03], pages 202–219.

[Joh97] Michael Johnson, editor. *Algebraic Methodology and Software Technology (Proceedings of AMAST '97, Sydney, Australia)*, volume 1349 of *Lecture Notes in Computer Science*. Springer-Verlag, December 1997.

[Jok98] Jyke Jokinen. Concurrent programming in Java is NOT the user-friendly way. `http://mail.python.org/pipermail/thread-sig/1998-February/000096.html`, 1998.

[JP01] Bart Jacobs and Eric Poll. A logic for the Java Modelling Language JML. In Hussmann [Hus01], pages 284–299.

[JR02] Bart Jacobs and Arend Rensink, editors. *Formal Methods for Open Object-Based Distributed Systems V, IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002), March 20-22*, volume 209. Kluwer, 2002.

[JvdBH+98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Barkum, Ulrich Hensel, and Hendrik Tews. Reasoning about classes in Java (preliminary report). In OOPSLA'98 [OOP98], pages 329–340. In *SIGPLAN Notices* 30(10).

[Kap92] Deepak Kapur, editor. *Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[KeY03] KeY: Integrated deductive software design. `http://i12www.ira.uka.de/~key/index.htm`, 2003.

[KG98] Lora L. Kassab and Steven J. Greenwald. Towards formalizing the Java security architecture of JDK 1.2. In *European Symposium on Research In Computer Security*, 1998.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Lam88] Leslie Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10(2):267–281, April 1988.

[Lam94] Leslie Lamport. Verification and specification of concurrent programs. In de Bakker et al. [dBdRR94], pages 347–374.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[Lan65] Peter J. Landin. A correspondence between Algol 60 and Church's lambda calculus. *Communications of the ACM*, 8(3):89–101; 158–165, 1965.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[LB99] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Java Technology*. Sun Microsystems Press Series. Pearson Education, 1999.

[LCC+03] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In Bonsangue et al. [BdBdRG03], pages 262–284.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*, volume 2. Addison-Wesley, 1999.

[LG81] Gary Levin and David Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.

[LM03] Hanbing Liu and J. Strother Moore. Executable JVM model for analytical reasoning: A study. In *Proc. of the ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators, San Diego, CA*, June 2003.

[Loo01] The LOOP project: Formal methods for object-oriented systems. `http://www.cs.kun.nl/\home{bart}/LOOP/`, 2001.

[LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. SRC Technical Note 1999-002, Compaq, May 1999.

[LW90] Gary T. Leavens and William E. Wheil. Reasoning about object-oriented programs that use subtypes. In OOPSLA'90 [OOP90], pages 212–223. Extended Abstract.

[LW95] Gary T. Leavens and William E. Wheil. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995. An expanded version appeared as Iowa State University Report, 92-28d.

[MAS00] Jan-Willem Maessen, Arvind, and Xiaowei Shen. Improving the Java memory model using CRF. In OOPSLA'00 [OOP00a]. In *SIGPLAN Notices*.

[McC65] John McCarthy. A formal description of a subset of ALGOL. In T.B. Steel Jr., editor, *Formal Language Description Languages*, pages 1–7, 1965.

[MD99] Ana M. D. Moreira and Serge Demeyer, editors. *Object-Oriented Technology, ECOOP'99 Workshop Reader, ECOOP'99 Workshops, Panels, and Posters, Lisbon, Portugal, June 14-18, 1999, Proceedings*, volume 1743 of *Lecture Notes in Computer Science*. Springer, 1999.

[Mét02] Daniel Le Métayer, editor. *Programming Languages and Systems: Proceedings of the 11th European Symposium on Programming (ESOP 2002), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2002), (Grenoble, France, April 8-12, 2002)*, volume 2305 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. Wiley, 1999.

[MKLP01] J. Strother Moore, Robert Krug, Hanbing Liu, and George Porter. Formal models of Java at the JVM level - A survey from the ACL2 perspective. In *Workshop on Formal Techniques for Java Programs*. June 2001.

[Möl02] Michael Möller. Specifying and checking Java using CSP. In *Proc. of the Workshop on Formal Techniques for Java-like Programs - FTfJP'2002*, 2002. Computing Science Department, University of Nijmegen, June 2002. Technical Report NIII-R0204.

[Moo99] J. Strother Moore. Proving theorems about Java-like byte code. In Olderog and Steffen [OS99], pages 139–162.

[Moo02] J. Strother Moore. Proving theorems about Java and the JVM with ACL2. In Broy and Pizka [BP02], pages 227–290.

[MP01a] Jeremy Manson and William Pugh. Core multithreaded semantics for Java. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2001 Conference, Stanford*, 2001.

[MP01b] Jeremy Manson and William Pugh. Semantics of multithreaded Java. Technical report, Dept. of Computer Science, University of Maryland, 2001.

[MP03] J. Strother Moore and George Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems*, 2003. submitted for publication.

[MY02] Tiziana Margaria and Wang Yi, editors. *Tools and Algorithms for the Construction and Analysis of Systems(TACAS' 02)*, volume 2031 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[NE02] Mogens Nielsen and Uffe H. Engberg, editors. *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2002), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2002), (Grenoble, France, April 8-12, 2002)*, volume 2303 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2002.

[Nip02] Tobias Nipkow. Hoare logics in Isabelle/HOL. In Schwichtenberg and Steinbrüggen [SS02], pages 341–367.

[NNS03] Elie Najm, Uwe Nestmann, and Perdita Stevens, editors. *Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '03), Paris*, volume 2884 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2003.

[NSH98] Elie Najm, Ahmed Serhrouchni, and Gerard Holzmann, editors. *Electronic Proceedings of the Fourth International SPIN Workshop, Paris, France*, November 1998.

[NvO98] Tobias Nipkow and David von Oheimb. Java-light is type-safe — definitely. In POPL'98 [POP98], pages 161–170.

[NvOP00] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

[OAdGBP00] Manuel Ojeda-Aciego, Inma P. de Guzmán, Gerhard Brewka, and Luís Moniz Pereira, editors. *Proceedings of the 8th European Workshop on Logics in AI (JELIA)*, volume 1919 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[OOP90] ACM. *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA) '90*, 1990. In *SIGPLAN Notices* 25(10).

[OOP98] ACM. *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA) '98 (Vancouver, Canada)*, 1998. In *SIGPLAN Notices* 30(10).

[OOP99] ACM. *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA) '99*, 1999. In *SIGPLAN Notices*.

[OOP00a] ACM. *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA) '00*, 2000. In *SIGPLAN Notices*.

[OOP00b] *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications European Conference on Object-Oriented Programming (OOPSLA) (ECOOP)*, 2000.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Kapur [Kap92], pages 748–752.

[OS99] Ernst-Rüdiger Olderog and Bernhard Steffen, editors. *Correct System Design - Recent Insights and Advances*, volume 1710 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[OSRSC99] Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Manual (Language Reference, Prover Guide, System Guide)*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[OW99] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly, Second edition, January 1999.

[Owi75] Susan Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, 1975.

[Pal00] Catuscia Palamidessi, editor. *CONCUR 2000: Concurrency Theory (11th International Conference, University Park, PA, USA)*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2000.

[Pat90] Michael S. Paterson, editor. *Seventeenth Colloquium on Automata, Languages and Programming (ICALP) (Warwick, England)*, volume 443 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Pau93] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

[PdB03] Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In Najm et al. [NNS03], pages 64–78. An extended version appeared as University of Utrecht Technical Report UU-CS-2003-010.

[PH97a] Arnd Poetzsch-Heffter. A logic for the verification of object-oriented programs. In Berghammer and Simon [BS97], pages 31–42. Bericht Nr. 9717.

[PH97b] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. Technische Universität München, January 1997. Habilitationsschrift.

[PHM98] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In Gries and de Roever [GdR98], pages 404–423.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In Swierstra [Swi99], pages 162–176.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

[POP98] ACM. *25th Annual Symposium on Principles of Programming Languages (POPL) (San Diego, CA)*, 1998.

[Pug00] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.

[PvdBJ00] Eric Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Acad. Publ., 2000.

[PvdBJ01] Eric Poll, Joachim van den Berg, and Bart Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[RHW01] Bernhard Reus, Rolf Hennicker, and Martin Wirsing. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In Hussmann [Hus01], pages 300–316.

[RKCW97] Bernhard Reus, Alexander Knapp, Pietro Cenciarelli, and Martin Wirsing. Verifying a compiler optimization for multi-threaded Java. In *Workshop on Algebraic Development Techniques*, pages 402–417, 1997.

[RM02] Abhik Roychoudhury and Tulika Mitra. Specifying multithreaded Java semantics for program verification. In *ACM/IEEE International Conference on Software Engineering (ICSE), pages 489-499.*, pages 489–499, 2002.

[Rus01] John M. Rushby. PVS bibliography, 2001. `http://www.csl.sri.com/papers/pvs-bib/`.

[RW00] Bernhard Reus and Martin Wirsing. A Hoare-logic for object-oriented programs. Technical report, LMU München, 2000.

[San01] David Sands, editor. *Programming Languages and Systems: Proceedings of the 10th European Symposium on Programming (ESOP 2001), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2001), (Genova, Italy, April 2001)*, volume 2028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. In *4th Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970.

[Sco76] Dana S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[SG94] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, 4th edition, 1994.

[SI298] *Proceedings of the Systems Implementation Conference (SI2000)*. Chapman & Hall, 1998.

[SS02] Helmut Schwichtenberg and Ralf Steinbrüggen, editors. *Proof and System-Reliability*. Kluwer, 2002.

[SS03] Robert F. Stärk and Joachim Schmid. Completeness of a bytecode verifier and a certifying Java-to-JVM compiler. *J. of Automated Reasoning*, 2003. Accepted for publication.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming LAnguage Semantics*. MIT Press, 1977.

[Swi99] S. Doaitse Swierstra, editor. *Proceedings of the 8th European Symposium on Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*. Springer, 1999.

[Sym97] Don Syme. Proving Java type soundness. Technical Report 427, Cambridge University, 1997.

[Sym99] Don Syme. Proving Java type soundness. In Alves-Foss [AF99], pages 83–118.

[TH02] Francis Tang and Martin Hofmann. Generation of verification conditions for Abadi and Leino's logic of objects (extended abstract). In FOOL2002 [FOO02]. A longer version is available as LFCS technical report.

[Tho99] Wolfgang Thomas, editor. *Proceedings of the Second International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '99), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'99), (Amsterdam, The Netherlands, April 1999)*, volume 1578 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[TZ88] John V. Tucker and Jeffery I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*, volume 6 of *CWI Monograph Series*. North-Holland, 1988.

[vdBHJP00] Joachim van den Berg, Marieke Huisman, Bart Jacobs, and Eric Poll. A type-theoretic memory model for verification of sequential Java programs. In Bert et al. [BCM00], pages 1–21. An earlier version appeared as Computer Science Institute, University of Nijmegen, Technical Report CSI-R9926, 1999.

[vdBJ02] Joachim van den Berg and Bart Jacobs. The Loop compiler for Java and JML. In Margaria and Yi [MY02], pages 299–312.

[vdBJP01] Joachim van den Berg, Bart Jacobs, and Eric Poll. Formal specification and verification of JavaCard's application identifier class. In Attali and Jensen [AJ01], pages 137–150.

[vO00a] David von Oheimb. Axiomatic semantics for Java^{light}. In OOPSLA2000 [OOP00b].

[vO00b] David von Oheimb. Axiomatic sematics for Java^{light} in Isabelle/HOL. Technical Report CSE 00-008, Oregon Graduate Institute, 2000.

[vO01] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[vON99] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Alves-Foss [AF99].

[vON02] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In Eriksson and Lindsay [EL02], pages 89–105.

[Wal97] Charles Wallace. The semantics of the Java programming language. Technical Report CSE-TR-355-97, University of Michigan, 1997.

[WK99] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml*. Object Technology Series. Addison-Wesley, 1999.

[YGL02] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Proc Joint ACM-ISCOPE Conference on Java*, pages 192–201, 2002.

Index

Notation index

The sequential language ($\boldsymbol{Java}_{seq}$)

Notation	Meaning	Page
c	class type	15
Bool	boolean type	15
Int	integer type	15
$t \times t$	tuple type	15
$\mathsf{list}\, t$	sequence type	15
f	operator	15
f	interpretation of operator f	15
$\dot\cup$	disjoint union operator	16
$x \in IVar$	instance variable	16
$u, v, \ldots \in TVar$	local variables	16
$y \in Var$	variable	16
this	self-reference	16
null	empty reference	16
$e \in Exp$	expression	16
e_{ret}	return expression	16
$stm \in Stm$	statement	16
ϵ	empty statement	16
$Meth_c$	set of methods of class c	17
$body_{m,c}$	body of method m of class c	17
$IVar_c$	set of instance variables of class c	17
Val^t	values of type t	19
$\alpha, \beta, \ldots \in Val^c$	object identifiers of type c	19
$null^c$	value of null^c	19
Val^c_{null}	$Val^c \,\dot\cup\, \{null^c\}$	19
Val	$\bigcup_t Val^t$	19
Val_{null}	$\bigcup_t Val^t_{null}$	19
$Init$	function assigning initial values to variables	19
$\tau \in \Sigma_{loc}$	local state	19
$\tau^{m,c}$	local state of method m of c	19

The concurrent language (*Java*$_{conc}$)

Reentrant monitors (*Java*$_{synch}$)

Notation	Meaning	Page
$wait(T, \alpha)$	the wait set of α	64
$notified(T, \alpha)$	the notified set of α	64
$signal(T, \alpha)$	set of threads after a !signal_all broadcast	66
$owns(T, \alpha)$	predicate expressing that a thread in T owns the lock of α	66
lock	built-in auxiliary instance variable (lock owner)	66
free	value of a free lock	66
wait	built-in auxiliary instance variable (wait set)	66
notified	built-in auxiliary instance variable (notified set)	66
inc(lock)	operator incrementing the lock value	66
dec(lock)	operator decrementing the lock value	66
$get(notified, \alpha)$	retrieves the value (α, n) from *notified*	67
$notify(wait, notified)$	represents the effect of notification	67

Weakest precondition calculus

Notation	Meaning	Page
$p[\vec{e}/\vec{y}]$	local substitution applied to p	75
$P[\vec{E}/z.\vec{x}]$, $P[\vec{E}/\vec{u}]$	global substitution applied to P	76

Soundness and completeness

Notation	Meaning	Page
φ_{prog}	the annotation of *prog*	81
$prog \models \varphi$	*prog* satisfies the specification φ	81
$prog' \vdash \varphi'$	$prog'$ with annotation φ' satisfies the verification conditions of the proof system	82
$\downarrow prog$	restriction operator	82
$proj(v, i)$	the ith component of the tuple v	83
$s[i]$	the ith element of the sequence s	84
$\models \varphi' \rightarrow \varphi$	the annotation φ' implies φ	84
loc	auxiliary local variable (program counter)	85

$l \equiv stm$	l represents the control point in front of stm	85
h_{inst}	auxiliary instance variable (local history)	86
h_{comm}	auxiliary instance variable (communication history)	86

Proving deadlock freedom

Notation	**Meaning**	**Page**
$I(z)$	$I[z/\mathsf{this}]$	92
$\mathsf{terminated}(z)$	global assertion expressing that the thread of z is terminated	92
$\mathsf{blocked}(z, z', p)$	global assertion expressing that the thread of z is disabled in the object z' at control point p	92
$\mathsf{blocked}(z, z')$	global assertion expressing that the thread of z is blocked in the object z'	92
$Ass(z')$	the set of all assertions in z'	92
$owns(thread, lock)$	*thread* owns *lock*	93
$not\_owns(thread, lock)$	*thread* does not own *lock*	93
$depth(lock)$	number of reentrant synchronized method invocations	93

Appendix A

Proofs

A.1 Properties of substitutions and projection

Proof A.1.1 (of Lemma 2.3.1) *By induction on the structure of local expressions and assertions. The base cases for local expressions are listed below, where the ones for instance and local variables are covered by the respective provisos of the lemma.*

$$
\begin{aligned}
[\![x[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![z.x]\!]_{\mathcal{G}}^{\omega,\sigma} = \sigma([\![z]\!]_{\mathcal{G}}^{\omega,\sigma})(x) = \sigma(\omega(z))(x) = [\![x]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \\
[\![u[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![u]\!]_{\mathcal{G}}^{\omega,\sigma} = \omega(u) = \tau(u) = [\![u]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \\
[\![\mathsf{this}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![z]\!]_{\mathcal{G}}^{\omega,\sigma} = \omega(z) = [\![\mathsf{this}]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \\
[\![\mathsf{null}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= \mathit{null} = [\![\mathsf{null}]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \\
[\![z'[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![z']\!]_{\mathcal{G}}^{\omega,\sigma} = \omega(z') = [\![z']\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \ .
\end{aligned}
$$

Compound expressions are treated by straightforward induction:

$$
\begin{aligned}
& [\![\mathsf{f}(e_1,\ldots,e_n)[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} & \\
= \ & f([\![e_1[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma},\ldots,[\![e_n[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma}) & \textit{semantics of assertions} \\
= \ & f([\![e_1]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau},\ldots,[\![e_n]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau}) & \textit{by induction} \\
= \ & [\![\mathsf{f}(e_1,\ldots,e_n)]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} & \textit{semantics of assertions}\ .
\end{aligned}
$$

For local assertions, negation and conjunction are straightforward. Unrestricted quantification $\exists z'.\ p$ in the local assertion language is only allowed for variables of type $t \in \{\mathsf{Int}, \mathsf{Bool}\}$ and for types composed from them, for which $Val^t_{null}(\sigma) = Val^t$. We get

$$
\begin{aligned}
& [\![(\exists z'.\ p)[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = \mathit{true} & \\
\iff\ & [\![\exists z'.\ p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = \mathit{true} & \textit{def. substitution} \\
\iff\ & [\![p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega[z'\mapsto v],\sigma} = \mathit{true\ for\ some}\ v \in Val^t & \textit{assertion semantics} \\
\iff\ & [\![p]\!]_{\mathcal{L}}^{\omega[z'\mapsto v],\sigma(\omega(z)),\tau} = \mathit{true\ for\ some}\ v \in Val^t & \textit{by induction} \\
\iff\ & [\![\exists z'.\ p]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} = \mathit{true} & \textit{assertion semantics.}
\end{aligned}
$$

For restricted quantification over elements of a sequence let $z' \in LVar^t$. Then

$$
\begin{array}{lll}
 & [\![(\exists z' \in e.\ p)[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = true & \\
\iff & [\![\exists z'.\ z' \in e[z/\mathsf{this}] \wedge p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = true & \textit{by definition} \\
\iff & [\![z' \in e[z/\mathsf{this}] \wedge p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma} = true & \textit{semantics} \\
 & \textit{for some } v \in Val^t_{null}(\sigma) \textit{ and } \omega' = \omega[z' \mapsto v] & \\
\iff & \left([\![z']\!]_{\mathcal{G}}^{\omega',\sigma} \in [\![e[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma} \wedge [\![p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma}\right) = true & \textit{semantics} \\
 & \textit{for some } v \in Val^t_{null}(\sigma) \textit{ and } \omega' = \omega[z' \mapsto v] & \\
\iff & \left([\![z']\!]_{\mathcal{L}}^{\omega',\sigma(\omega(z)),\tau} \in [\![e]\!]_{\mathcal{L}}^{\omega',\sigma(\omega(z)),\tau} \wedge [\![p]\!]_{\mathcal{L}}^{\omega',\sigma(\omega(z)),\tau}\right) = true & \textit{by induction} \\
 & \textit{for some } v \in Val^t_{null}(\sigma) \textit{ and } \omega' = \omega[z' \mapsto v] & \\
\iff & [\![(z' \in e) \wedge p]\!]_{\mathcal{L}}^{\omega',\sigma(\omega(z)),\tau} = true & \textit{semantics} \\
 & \textit{for some } v \in Val^t_{null}(\sigma) \textit{ and } \omega' = \omega[z' \mapsto v] & \\
\iff & [\![\exists z' \in e.\ p]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} = true & \textit{semantics}\ .
\end{array}
$$

The last step uses the assumption that the local state τ and the instance state $\sigma(\omega(z))$ assign values from $Val_{null}(\sigma)$ to all variables, i.e., e does not refer to values of non-existing objects (see Lemma A.1.4). Consequently, $v \in Val^t_{null}$ together with $[\![z' \in e]\!]_{\mathcal{L}}^{\omega[z' \mapsto v],\sigma(\omega(z)),\tau} = true$ implies $v \in Val^t_{null}(\sigma)$. The case for restricted quantification over subsequences is analogous. □

Proof A.1.2 (of Lemma 5.1.1) *We proceed by straightforward induction on the structure of local assertions. Let $\acute{\sigma}_{inst} = \grave{\sigma}_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}}]$ and $\acute{\tau} = \grave{\tau}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}}]$. In the case for local variables $u = y_i$ we get*

$$
\begin{array}{rcl}
[\![u[\vec{e}/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}} & = & [\![e_i]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}} \\
 & = & \acute{\tau}(u) \\
 & = & [\![u]\!]_{\mathcal{L}}^{\omega,\acute{\sigma}_{inst},\acute{\tau}}\ .
\end{array}
$$

For instance variables $x = y_i$ similarly:

$$
\begin{array}{rcl}
[\![x[\vec{e}/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}} & = & [\![e_i]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}} \\
 & = & \acute{\sigma}_{inst}(x) \\
 & = & [\![x]\!]_{\mathcal{L}}^{\omega,\acute{\sigma}_{inst},\acute{\tau}}\ .
\end{array}
$$

The remaining cases are straightforward. □

Proof A.1.3 (of Lemma 5.1.2) *Let $\acute{\omega} = \grave{\omega}[\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}]$ and let $\acute{\sigma}$ be defined by $\grave{\sigma}[[\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}.\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}]$. We proceed by induction on the structure of global expressions and assertions. The base cases for* null *and z' are straightforward. For the induction cases, we start with the crucial one for qualified reference to* *instance variables. For expressions $E'.x[\vec{E}/z.\vec{y}]$ with x not in $\vec{y}$ the property holds by induction. So assume that x is in $\vec{y}$:*

$$[\![(E'.y_i)[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} \quad = \quad [\![\mathsf{if}\ E'[\vec{E}/z.\vec{y}] = z\ \mathsf{then}\ E_i\ \mathsf{else}\ (E'[\vec{E}/z.\vec{y}]).y_i\ \mathsf{fi}]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} \ .$$

This conditional assertion evaluates to $[\![E_i]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}$ if $[\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = [\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}$ and to $[\![(E'[\vec{E}/z.\vec{y}]).y_i]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}$ otherwise. So in the first case we get

$$\begin{array}{rcll}
[\![(E'.y_i)[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & = & [\![E_i]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & \\
& = & \acute{\sigma}([\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(y_i) & \textit{by def. of } \acute{\sigma} \\
& = & \acute{\sigma}([\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(y_i) & \textit{by the case assumption} \\
& = & \acute{\sigma}([\![E']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}})(y_i) & \textit{by induction} \\
& = & [\![E'.y_i]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} & \textit{by def. of } [\![_]\!]_{\mathcal{G}} \ .
\end{array}$$

If otherwise $[\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} \neq [\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}$, then

$$\begin{array}{rcll}
[\![(E'.y_i)[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & = & [\![(E'[\vec{E}/z.\vec{y}]).y_i]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & \\
& = & \grave{\sigma}([\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(y_i) & \textit{by def. of } [\![_]\!]_{\mathcal{G}} \\
& = & \acute{\sigma}([\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(y_i) & \textit{case assumption+def. } \acute{\sigma} \\
& = & \acute{\sigma}([\![E']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}})(y_i) & \textit{by induction} \\
& = & [\![E'.y_i]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} & \textit{by def. of } [\![_]\!]_{\mathcal{G}} \ .
\end{array}$$

For operator expressions we get:

$$\begin{array}{rll}
& [\![(\mathsf{f}(E_1,\ldots,E_n))[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & \\
= & [\![\mathsf{f}(E_1[\vec{E}/z.\vec{y}],\ldots,E_n[\vec{E}/z.\vec{y}])]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & \textit{def. substitution} \\
= & f([\![E_1[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}},\ldots,[\![E_n[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}) & \textit{def. } [\![_]\!]_{\mathcal{G}} \\
= & f([\![E_1]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}},\ldots,[\![E_n]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}}) & \textit{by induction} \\
= & [\![\mathsf{f}(E_1,\ldots,E_n)]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} & \textit{def. } [\![_]\!]_{\mathcal{G}} \ .
\end{array}$$

For global assertions, the cases of negation and conjunction are straightforward. For quantification,

$$\begin{array}{rll}
& [\![(\exists z'.\ P)[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = \textit{true} & \\
\iff & [\![\exists z'.\ P[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = \textit{true} & \textit{def. substitution} \\
\iff & [\![P[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega}[z' \mapsto v],\grave{\sigma}} = \textit{true for some } v \in \textit{Val}_{\textit{null}}(\grave{\sigma}) & \textit{def. } [\![_]\!]_{\mathcal{G}} \\
\iff & [\![P]\!]_{\mathcal{G}}^{\acute{\omega}[z' \mapsto v],\acute{\sigma}} = \textit{true for some } v \in \textit{Val}_{\textit{null}}(\grave{\sigma}) & \textit{by induction} \\
\iff & [\![\exists z'.\ P]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} = \textit{true}\ , & \textit{Val}(\grave{\sigma}) = \textit{Val}(\acute{\sigma})
\end{array}$$

where z' is not in $\vec{y}$ (otherwise the substitution renames z'). □

Lemma A.1.4 *Let σ be a global state and ω a logical environment referring only to values existing in σ. Then $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in \textit{Val}_{\textit{null}}(\sigma)$ for all global expressions $E \in \textit{GExp}$ that can be evaluated in the context of ω and σ.*

Proof A.1.5 (of Lemma A.1.4) *By structural induction on the global assertion. The case for logical variables $z \in LVar^t$ is immediate by the assumption about ω, the ones for* null *and operator expressions are trivial, respectively follows by induction. For qualified references $E.x$ with E a global expression of type c and x an instance variable of type t in class c, if $E.x$ can be evaluated in the context of ω and σ, then $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \neq null$. Hence by induction $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in Val_{null}(\sigma)$, more specifically $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in Val(\sigma)$. Therefore by definition of global states $\sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) \in Val_{null}(\sigma)$.* $\square$

Proof A.1.6 (of Lemma 2.4.18) *We prove the lemma by structural induction on global assertions. Assume a global state $\grave{\sigma}$, and let $\acute{\sigma} = \grave{\sigma}[\alpha \mapsto \sigma_{inst}^{c,init}]$ be an extension of $\grave{\sigma}$ with a new object $\alpha \in Val^c$, $\alpha \notin Val(\grave{\sigma})$. Assume furthermore a logical environment ω referring only to values existing in $\grave{\sigma}$, and let v be the sequence consisting of all elements of $\bigcup_c Val_{null}^{c}(\grave{\sigma})$. Let finally P be a global assertion, $z' \in LVar^{\mathsf{list\,Object}}$ a logical variable not occurring in P, and $\acute{\omega} = \grave{\omega}[z' \mapsto v]$. Since z' is fresh in P, we have for all logical variables z in P that $[\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = \grave{\omega}(z) = \acute{\omega}(z) = [\![z]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} = [\![z \downarrow z']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}}$. For qualified references to instance variables, the argument is as follows:*

$$
\begin{array}{rcll}
[\![E.x]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & = & \grave{\sigma}([\![E]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(x) & \textit{semantics} \\
 & = & \acute{\sigma}([\![E]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(x) & [\![E]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} \neq \alpha \textit{ by Lemma A.1.4 and } \alpha \notin Val(\grave{\sigma}) \\
 & = & \acute{\sigma}([\![E \downarrow z']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}})(x) & \textit{by induction} \\
 & = & [\![(E \downarrow z').x]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} & \textit{semantics} \\
 & = & [\![(E.x) \downarrow z']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} & \textit{def. } \downarrow z' .
\end{array}
$$

The interesting case is the one for quantification. For $z \in LVar^t$:

$$
\begin{array}{crcll}
 & \grave{\omega}, \grave{\sigma} & \models_{\mathcal{G}} & \exists z.\ P & \\
\Longleftrightarrow & \grave{\omega}[z \mapsto u], \grave{\sigma} & \models_{\mathcal{G}} & P \textit{ for some } u \in Val_{null}^{t}(\grave{\sigma}) & \textit{semantics} \\
\Longleftrightarrow & \acute{\omega}[z \mapsto u], \acute{\sigma} & \models_{\mathcal{G}} & P \downarrow z' \textit{ for some } u \in Val_{null}^{t}(\grave{\sigma}) & \textit{induction} \\
\Longleftrightarrow & \acute{\omega}[z \mapsto u], \acute{\sigma} & \models_{\mathcal{G}} & \mathsf{obj}(z) \subseteq z' \wedge P \downarrow z' & obj(u) \subseteq v \\
 & & & \quad \textit{for some } u \in Val_{null}^{t}(\grave{\sigma}) & \\
\Longleftrightarrow & \acute{\omega}, \acute{\sigma} & \models_{\mathcal{G}} & \exists z.\ \mathsf{obj}(z) \subseteq z' \wedge P \downarrow z' & \textit{semantics} \\
\Longleftrightarrow & \acute{\omega}, \acute{\sigma} & \models_{\mathcal{G}} & (\exists z.\ P) \downarrow z'. &
\end{array}
$$

The remaining cases are straightforward. $\square$

A.2 Soundness

This section contains the inductive proof of soundness of the proof method. We start with some ancillary lemmas about basic invariant properties of proof outlines, for instance properties of the built-in auxiliary variables added in the transformation. Afterwards, we show soundness of the proof system.

A.2.1 Invariant properties

Proof A.2.1 (of the transformation Lemma 6.1.1) *We proceed for both directions by straightforward induction on the length of reduction. The only interesting property of the transformation is the representation of notification by a single auxiliary assignment of the notifier. For this case we use Lemma 6.1.3 showing soundness of the representation of the wait and notified sets by the auxiliary instance variables* wait *and* notified. $\square$

Proof A.2.2 (of Lemma 6.1.2) *All parts by straightforward induction on the length of computations.* $\square$

Proof A.2.3 (of Lemma 6.1.3) *If the order of the elements is unimportant, in the following we also use set notation for the values of the* wait *and* notified *variables. Correctness of the projection operation uses the results of this lemma and is formulated in Lemma 6.1.1.*

The cases 2a and 2b are satisfied by the definition of the projection operator. Inductivity for the cases 2c and 2d are easy to show using Lemma 6.1.2 and the cases 2a and 2b of this lemma. For the other cases we proceed by induction on the length of the run $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \acute{T}, \acute{\sigma} \rangle$ *of the proof outline* $prog'$.

In the base case of an initial configuration $\langle T_0, \sigma_0 \rangle$ *(cf. page 20), the set* T_0 *contains exactly one thread* (α, τ, stm)*, executing the non-synchronized main-statement of the program, i.e.,* $\neg owns(T_0 \downarrow prog, \alpha)$*, and initially the lock of the only object* α *is set to free. Furthermore, the instance variables* wait *and* notified *of the initial object are set to* $\emptyset$*, and the wait and notified sets of the semantics are also empty.*

For the inductive step, assume $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$*. We distinguish on the kind of the last computation step.*

Case: CALL_{start}, $\text{CALL}^{skip}_{start}$, RETURN_{run}

In these cases none of the concerned variables or predicates are touched, and the property follows directly by induction.

Case: ASS_{inst}, ASS_{loc}

Note that this case handles assignments, but not the observations of communication and object creation. Remember furthermore that the signaling mechanism is implemented in proof outlines by auxiliary assignments, and thus this case covers also the rules SIGNAL, SIGNAL_{skip}*, and* SIGNALALL.

If the assignment is not in a notify *or in a* notifyAll *method representing notification, then the case is analogous to the above one.*

Assume first that the assignment in the last computation step represents notification in a notify *method of the proof outline, and that the wait set is not empty. I.e., a thread* $\xi_1 \in \grave{T}$ *notifies another thread* $\xi_2 = (\alpha_2, \tau, stm) \circ \xi_2' \in \grave{T}$ *in the wait set of* α*. Remember that notification is represented by a single assignment of the notifier, and thus the stack of the notified thread* ξ_2 *does not change. However, according to the projection definition, as the notifier changes* *the value of* wait *of* α*, the projection* $\xi_2 \downarrow prog$ *represents a thread being in the wait set in* $\langle \grave{T}, \grave{\sigma} \rangle$ *and being in the notified set in* $\langle \acute{T}, \acute{\sigma} \rangle$*.*

The only relevant effect of the step is moving $(\alpha_2, n) \in \grave{\sigma}(\alpha)(\mathsf{wait})$ *from the wait set into the notified set of* α*, where* n *is by induction the number of synchronized invocations of* ξ_2 *in* α*. Thus the properties 1a, 1b and 2e are automatically invariant. Induction implies also uniqueness of the representation of the wait and notified sets, i.e.,* α_2 *is contained neither in* $\grave{\sigma}(\alpha)(\mathsf{notified})$ *nor in* $\acute{\sigma}(\alpha)(\mathsf{wait})$*. Thus moving the thread of* α_2 *from the wait into the notified set does not violate uniqueness of the representation.*

If the wait set $\grave{\sigma}(\alpha)(\mathsf{wait})$ *is empty, then no notification takes place; the property follows directly by induction.*

The case for the assignment in the notifyAll *method is analogous, with the difference that all threads in the wait set get notified by* ξ_1*. The notifier observation sets the value of the auxiliary instance variable* notified *of* α *to* $\grave{\sigma}(\alpha)(\mathsf{notified}) \,\dot{\cup}\, \grave{\sigma}(\alpha)(\mathsf{wait})$*, whereas the corresponding* wait *variable gets the value* $\emptyset$*. By induction we have* $\grave{\sigma}(\alpha)(\mathsf{notified}) \cap \grave{\sigma}(\alpha)(\mathsf{wait}) = \emptyset$*, and thus the required properties are invariant under notification.*

Case: NEW
Assume that the last step creates a new object, and executes the corresponding observation. Let $\alpha \in dom(\acute{\sigma})$*. Then* α *either references the newly created object, or* $\alpha \in dom(\grave{\sigma})$*. In the first case* $\alpha \notin dom(\grave{\sigma})$*, and by the definition of global configurations (cf. page 19) there is no local configuration* $(\alpha, \tau, stm) \in \grave{T}$*, and the wait and notified set of* α *in* $\grave{T}$ *are empty. Since the last step does not add any local configurations to* $\grave{T}$*, we have* $\alpha \neq \beta$ *for all* $(\beta, \tau, stm) \in \acute{T}$ *and thus* $\neg owns(\acute{T} \downarrow prog, \alpha)$*. Since the lock of the new object is initialized to free, and* wait *and* notified *of* α *get the value* $\emptyset$*, the required property holds for the new object. In the second case, if* $\alpha \in dom(\grave{\sigma})$*, the property follows directly by induction.*

Case: CALL
Let $\alpha \in dom(\acute{\sigma})$*. Then also* $\alpha \in dom(\grave{\sigma})$*. If* α *is not the callee object, then the property holds directly by induction. If* α *is the callee object, the only new local configuration* (α, τ, stm) *in* $\acute{T}$ *represents the execution of the invoked method.*

If the invoked method is non-synchronized, then the property follows by induction (invocations of monitor methods are covered by the $\text{CALL}_{monitor}$ *case below). In the case of a synchronized method, let* $\xi \in \grave{T}$ *be the executing thread. The antecedent* $\neg owns(\grave{T} \setminus \{\xi\} \downarrow prog, \alpha)$ *implies by induction that, if there is no local configuration in* ξ *which executes a synchronized method of* α *then* $\grave{\sigma}(\alpha)(\mathsf{lock}) = free$*, and* $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$ *otherwise, where* $(\alpha_0, \tau_0, stm_0)$ *is the deepest configuration in* ξ *and* n *is the number of local configurations in* ξ *which execute synchronized methods of* α*. If in the state prior to the method invocation* $\grave{\sigma}(\alpha)(\mathsf{lock}) = free$*, then* (α, τ, stm) *is the only local configuration in* $\acute{T}$ *representing the execution of a synchronized method of* α *by a thread not in the wait or notified sets of* α*. Furthermore, the callee observation sets* $\acute{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, 1)$*, and thus the required property holds. In the second case, using the fact that the callee configuration is on top of its stack, the callee* *observation changes* $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$ *to* $\acute{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n+1)$*, and we get the property by Lemma 6.1.2 and by induction.*

Case: $\textsc{Call}_{monitor}$
Similarly to the case $\textsc{Call}$*, for* $\alpha \in dom(\acute{\sigma})$ *also* $\alpha \in dom(\grave{\sigma})$*, and if* α *is not the callee object, then the property holds by induction. In the case of the non-synchronized* notify *and* notifyAll *methods, none of the concerned variables or predicates are touched, and thus the property holds by induction again. So let* $\xi \in \grave{T}$ *be the executing thread invoking the non-synchronized* wait *method of* α*.*

The antecedent $owns(\xi \downarrow prog, \alpha)$ *implies by induction* $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$*, where* $(\alpha_0, \tau_0, stm_0)$ *is the deepest configuration in the stack* ξ *and* n *is the number of its synchronized method invocations in* α*. Furthermore, since* ξ *does not yet execute a* wait *method prior to the call, from* $\xi \notin wait(\grave{T} \downarrow prog, \alpha) \cup notified(\grave{T} \downarrow prog, \alpha)$ *we conclude by induction that* α_0 *is contained neither in* wait *nor in* notified *of* α *in* $\grave{\sigma}$*.*

The execution places the thread into α*'s wait set and, since at most one thread can own a lock at a time, it gives the lock of* α *free, i.e., we have* $\neg owns(\acute{T} \downarrow prog, \alpha)$*. The corresponding callee observation extends* $\grave{\sigma}(\alpha)(\mathsf{wait})$ *with* (α_0, n)*, and sets the lock-value of* α *to free. Thus the case follows by induction.*

Case: $\textsc{Return}$
Assume $\alpha \in dom(\acute{\sigma}) = dom(\grave{\sigma})$*. If* α *is not the callee object, or if the invoked method is non-synchronized, then the property holds directly by induction. Note that returning from the* wait *method is covered by the* $\textsc{Return}_{wait}$ *case below. So let* $\xi \in \grave{T}$ *be the thread of* α_0 *returning from a synchronized method of* α*; we denote the thread after execution by* $\xi' \in \acute{T}$*.*

Since ξ *is neither in the wait nor in the notified set of* α*, we get by definition* $owns(\xi \downarrow prog, \alpha)$ *prior to execution. If the given method is the only synchronized method of* α *executed by* ξ*, then in the successor configuration* $\neg owns(\xi' \downarrow prog, \alpha)$*, and from the invariant property that at most one thread can own a lock at a time we imply* $\neg owns(\acute{T} \downarrow prog, \alpha)$*. Otherwise, if* ξ *has reentrant synchronized method invocations in* α*, then the thread does not give the lock free upon return, i.e., in the successor state we still have* $owns(\xi' \downarrow prog, \alpha)$*.*

Using $owns(\xi \downarrow prog, \alpha)$*, we get by induction* $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$*, where* n *is the number of invocations of synchronized methods of* α *by* ξ*. The auxiliary variable* lock *of* α *is set by the callee augmentation to free, if* $n = 1$*, and to* $(\alpha_0, n-1)$*, otherwise. Since the auxiliary variables* wait *and* notified *are not touched, the property follows by induction.*

Case: $\textsc{Return}_{wait}$
Assume that the thread $\xi \in \grave{T}$ *of an object* α_0 *is returning from the* wait *method of* $\alpha \in dom(\acute{\sigma}) = dom(\grave{\sigma})$*; we denote the thread after execution by* $\xi' \in \acute{T}$*.*

The semantics assures $\neg owns(\grave{T} \downarrow prog, \alpha)$ *and by definition* $\xi \in notified(\grave{T} \downarrow prog, \alpha)$*. We get by induction* $\grave{\sigma}(\alpha)(\mathsf{lock}) = free$ *and* $(\alpha_0, n) \in \grave{\sigma}(\alpha)(\mathsf{notified})$*, where* n *is the number of invocations of synchronized methods of* α *by* ξ*. After returning, the thread gets removed from the notified set of* α *and gathers the lock of* α*, i.e.,* $\xi' \notin notified(\acute{T} \downarrow prog, \alpha)$ *and* $owns(\xi' \downarrow prog, \alpha)$*.*

The augmentation of the wait *method removes* (α_0, n) *from* $\grave{\sigma}(\alpha)(\mathsf{notified})$*; from the uniqueness of the representation follows* $\alpha_0 \neq \beta$ *for all* $(\beta, m) \in \acute{\sigma}(\alpha)(\mathsf{notified})$*. Furthermore, the observation sets the lock of* α *to* (α_0, n)*, by which we get the required property.* □

Proof A.2.4 (of Lemma 6.1.4) *Straightforward by the definition of augmentation.* □

A.2.2 Proof of the soundness theorem

Proof A.2.5 (of the soundness Theorem 6.1.5) *We prove the theorem by induction on the length of the computation, simultaneously for all parts of Definition 6.0.1.*

For the initial case let $dom(\sigma_0) = \{\alpha\}$*,* $\sigma_0(\alpha) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \alpha]$*,* $\tau_0 = \tau_{init}[\mathsf{thread} \mapsto \alpha]$*, and let* $\{p_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{p_3\}$ *stm be the main statement. Then the initial configuration* $\langle T_0', \sigma_0'\rangle$ *of the proof outline satisfies the following:* $\sigma_0' = \sigma_0[\alpha.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_0(\alpha),\tau_0}]$*, and for the stack we have* $T_0' = \{(\alpha, \tau_0', stm)\}$ *with* $\tau_0' = \tau_0[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_0(\alpha),\tau_0}]$*.*

Let ω *be a logical environment referring only to values existing in* σ_0*. As in* σ_0 *there exists exactly one object* α *being in its initial instance state, we have*

$$\omega[z \mapsto \alpha], \sigma_0 \quad \models_{\mathcal{G}} \quad \mathsf{InitState}(z) \wedge \forall z'.\ z'{=}\mathsf{null} \vee z{=}z'\ ,$$

where z *is of the type of the main class, and* z' *is a logical variable of type* Object*. Using the initial correctness condition we get*

$$\omega[z \mapsto \alpha], \sigma_0 \quad \models_{\mathcal{G}} \quad (GI \wedge P_3(z) \wedge I(z)) \circ f_{obs} \circ f_{init}$$

with I *the class invariant of* α*,* $\vec{v}$ *the local variables of the* run *method of the main class, and*

$$\begin{array}{rcl} f_{init} & = & [\mathsf{this}, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]\ , \ \textit{and} \\ f_{obs} & = & [\vec{E}_2(z)/z.\vec{y}_2]\,. \end{array}$$

Applying Lemma 5.1.2, we get for the global invariant $\omega', \sigma_0' \models_{\mathcal{G}} GI$ *for* $\omega' = \omega[z \mapsto \alpha][\vec{v} \mapsto \tau_0'(\vec{v})]$*. Since* GI *may not contain free logical variables, its value does not depend on the logical environment, and therefore* $\omega, \sigma_0' \models_{\mathcal{G}} GI$*.*

Similarly for the local property p_3*, we get with Lemma 5.1.2 that* $\omega', \sigma_0' \models_{\mathcal{G}} P_3(z)$*. With Lemma 2.3.1 we get* $\omega', \sigma_0'(\alpha), \tau_0' \models_{\mathcal{L}} pre(stm)$*. Since* $pre(stm)$ *does not contain free logical variables, we get finally* $\omega, \sigma_0'(\alpha), \tau_0' \models_{\mathcal{L}} pre(stm)$*. Part 3 for the class invariant is analogous.*

For the inductive step, assume $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma}\rangle \longrightarrow \langle \acute{T}, \acute{\sigma}\rangle$ *such that* $\langle \grave{T}, \grave{\sigma}\rangle$ *satisfies the conditions of Definition 6.0.1. Let* ω *be a logical environment referring only to values existing in* $\acute{\sigma}$*. We distinguish on the kind of the computation step* $\langle \grave{T}, \grave{\sigma}\rangle \longrightarrow \langle \acute{T}, \acute{\sigma}\rangle$*.*

If the computation step is executed by a single local configuration, we use the local correctness conditions for inductivity of the executing local configuration's properties, and the interference freedom test for all other local configurations and the class invariants in $\langle \acute{T}, \acute{\sigma} \rangle$*. For communication, invariance for the executing partners and the global invariant is shown using the cooperation test for communication. Communication itself does not affect the global state; invariance of the remaining properties under the corresponding observations is shown again with the help of the interference freedom test. Finally for object creation, invariance for the global invariant, the creator local configuration, the created object's class invariant is assured by the conditions of the cooperation test for object creation; all other properties are shown to be invariant using the interference freedom test.*

Case: ASS_{inst}, ASS_{loc}
Note that signaling is represented in proof outlines by auxiliary assignments, thus this case covers also the rules SIGNAL, SIGNALALL, *and* SIGNAL_{skip}. *Note furthermore that this case does not cover observations of communication or object creation.*

Let the last computation step be the execution of an assignment in the local configuration $(\alpha, \grave{\tau}_1, \vec{y} := \vec{e}; stm_1) \in \grave{T}$ *resulting in* $(\alpha, \acute{\tau}_1, stm_1) \in \acute{T}$*. According to the semantics,* $\acute{\tau}_1 = \grave{\tau}_1[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$ *and* $\acute{\sigma} = \grave{\sigma}[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$.

Since assignments, that does not observe object creation or communication, must not change the values of variables occurring in GI*, part (2) is satisfied.*

For part (1), assume $(\beta, \tau_2, stm_2) \in \acute{T}$*. If* $(\beta, \tau_2, stm_2) = (\alpha, \acute{\tau}_1, stm_1)$ *is the executing local configuration, then by induction* $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} pre(\vec{y} := \vec{e}) \wedge I$*, where* I *is the class invariant of* α*. The local correctness condition implies that* $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} pre(stm_1)[\vec{e}/\vec{y}]$*. Using the properties of the local substitution formulated in Lemma 5.1.1 we get* $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} pre(stm_1)$.

If otherwise (β, τ_2, stm_2) *is not the executing local configuration, then it is contained in* $\grave{T}$*. If* $\alpha \neq \beta$*, i.e., the execution does not take place in* β*, then* $\grave{\sigma}(\beta) = \acute{\sigma}(\beta)$*, and thus* $\omega, \acute{\sigma}(\beta), \tau_2 \models_{\mathcal{L}} pre(stm_2)$ *by induction. Otherwise let* τ *be* $\grave{\tau}_1[\vec{v}' \mapsto \tau_2(\vec{v})]$*, where* $\vec{v} = dom(\tau_2)$ *and* $\vec{v}'$ *fresh. Then Lemma 6.1.2, the induction assumptions, and the definition of* interferes *imply*

$$\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e}) \wedge pre'(stm_2) \wedge I \wedge \mathsf{interferes}(pre(stm_2), \vec{y} := \vec{e}) ,$$

and with the interference freedom test we get $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_2)[\vec{e}/\vec{y}]$*. Using the substitution Lemma 5.1.1 and the fact that, due to the renaming mechanism, no variables in* $\vec{v}'$ *may occur in* $\vec{y}$*, yields* $\omega, \acute{\sigma}(\alpha), \tau_2 \models_{\mathcal{L}} pre(stm_2)$.

Part (3) is similar, using the fact that the class invariant may contain instance variables only, and thus its evaluation does not depend on the local state.

Case: CALL
Let $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1) \in \grave{T}$ *be the caller configuration prior to method invocation, and let* $(\alpha, \acute{\tau}_1, stm_1') \in \acute{T}$ *and* $(\beta, \acute{\tau}_2, stm_2) \in \acute{T}$ *be the local configurations of the caller and the callee after execution. Let furthermore* $\langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} stm_2$ *be the invoked method's body and* $\vec{u}$ *its formal parameters. Directly after communication the callee has the local state* $\hat{\tau}_2 = \tau_{init}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$;

after the caller observation, the global state is $\hat{\sigma} = \grave{\sigma}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}]$ *and the caller's local state is updated to* $\acute{\tau}_1 = \grave{\tau}_1[\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}]$. *Finally, the callee observation updates its local state to* $\acute{\tau}_2 = \hat{\tau}_2[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta),\hat{\tau}_2}]$ *and the global state to* $\acute{\sigma} = \hat{\sigma}[\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta),\hat{\tau}_2}]$. *Let* $\vec{v}_1$ *denote* $dom(\grave{\tau}_1)$ *and assume* $\grave{\omega} = \omega[z \mapsto \alpha][z' \mapsto \beta][\vec{v}_1 \mapsto \grave{\tau}_1(\vec{v}_1)]$.

The semantics assures $\alpha \neq null$ *and* $\beta = [\![e_0]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1} \neq null$, *and we get with Lemma 2.3.1 and the definition of* $\grave{\omega}$ *that* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z \neq \mathsf{null} \wedge z' \neq \mathsf{null} \wedge E_0(z) = z'$.

If the method is synchronized and ξ *is the stack of the executing thread in* $\grave{T}$, *then according to the transition rule* $\neg owns(\grave{T} \backslash \{\xi\} \downarrow prog, \beta)$. *Using Lemma 6.1.3 and Lemma 6.1.2 we get* $\grave{\sigma}(\beta)(\mathsf{lock}) = \mathit{free} \vee \mathit{thread}(\grave{\sigma}(\beta)(\mathsf{lock})) = \grave{\tau}_1(\mathsf{thread})$ *and thus* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$.

In the following let $p_1 = pre(u_{ret} := e_0.m(\vec{e}))$, $p_2 = pre(\vec{y}_1 := \vec{e}_1)$, $p_3 = post(\vec{y}_1 := \vec{e}_1)$, $q_1 = I_q$, $q_2 = pre(\vec{y}_2 := \vec{e}_2)$, *and* $q_3 = post(\vec{y}_2 := \vec{e}_2)$, *where* I_q *is the class invariant of the callee. Let* I_p *be the caller's class invariant. Then we have by induction* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI$, *for the class invariants* $\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} I_p$ *and* $\grave{\omega}, \grave{\sigma}(\beta), \grave{\tau}_1 \models_{\mathcal{L}} I_q$, *and for the precondition of the call* $\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_1$. *Using the lifting lemma, the cooperation test for communication implies*

$$\grave{\omega}, \grave{\sigma} \quad \models_{\mathcal{G}} \quad (GI \wedge P_3(z) \wedge Q_3'(z'))[\vec{E}_2'(z')/z'.\vec{y}_2'][\vec{E}_1(z)/z.\vec{y}_1][\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}'] \ ,$$

where $\vec{v}$ *contains the local variables of the callee without the formal parameters* $\vec{u}$. *Using the lifting lemma again but in the reverse direction and Lemma 5.1.2 results* $\omega, \acute{\sigma} \models_{\mathcal{G}} GI$, *and thus part (2). Note that in the annotation no free logical variables occur, and thus the values of assertions in a proof outline do not depend on the logical environment. Furthermore, using the same lemmas we get*

$$\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3 \quad \text{and} \quad \omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} q_3 \ .$$

Thus part (1) is satisfied for the local configurations involved in the last computation step. All other configurations (γ, τ_3, stm_3) *in* $\acute{T}$ *are also in* $\grave{T}$. *If* $\gamma \neq \alpha$ *and* $\gamma \neq \beta$, *then* $\grave{\sigma}(\gamma) = \acute{\sigma}(\gamma)$, *and thus* $\omega, \acute{\sigma}(\gamma), \tau_3 \models_{\mathcal{L}} pre(stm_3)$ *by induction.*

Assume next $\gamma = \alpha$ *and* $\alpha \neq \beta$, *and let* τ *be* $\grave{\tau}_1[\vec{v}' \mapsto \tau_3(\vec{v})]$, *where* $\vec{v} = dom(\tau_3)$. *Then Lemma 6.1.2, the induction assumptions, and the definition of the assertion* interferes *imply with the interference freedom test* $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_3)[\vec{e}_1/\vec{y}_1]$. *The substitution Lemma 5.1.1 and the fact that, due to the renaming mechanism, no local variables in* $\vec{v}'$ *occur in* $\vec{y}_1$, *yield* $\omega, \hat{\sigma}(\alpha), \tau_3 \models_{\mathcal{L}} pre(stm_3)$. *Now, since* $\beta \neq \alpha$, *the callee observation also does not change the caller's instance state, and we have* $\hat{\sigma}(\alpha) = \acute{\sigma}(\alpha)$. *Thus we get* $\omega, \acute{\sigma}(\alpha), \tau_3 \models_{\mathcal{L}} pre(stm_3)$.

The case $\gamma = \beta$ *and* $\alpha \neq \beta$ *is similar. Communication and caller observation do not change the instance state of* β, *i.e.,* $\grave{\sigma}(\beta) = \hat{\sigma}(\beta)$. *The interference freedom test results* $\omega, \hat{\sigma}(\beta), \tau \models_{\mathcal{L}} pre'(stm_3)[\vec{e}_2/\vec{y}_2]$ *with* $\tau = \hat{\tau}_2[\vec{v}' \mapsto \tau_3(\vec{v})]$. *Due to the renaming mechanism, we conclude with the local substitution lemma that* $\omega, \acute{\sigma}(\beta), \acute{\tau} \models_{\mathcal{L}} pre'(stm_3)$ *with* $\acute{\tau}(\vec{v}') = \tau_3(\vec{v})$, *and thus* $\omega, \acute{\sigma}(\beta), \tau_3 \models_{\mathcal{L}} pre(stm_3)$.

For the last case $\gamma = \alpha = \beta$ note that, according to the restrictions on the augmentation, the caller may not change the instance state. Thus the same arguments as for $\gamma = \beta$ and $\alpha \neq \beta$ apply. I.e., part (1) is satisfied.

Part (3) is analogous: The interference freedom test implies $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} I_p$. Since I_p may contain instance variables only, its evaluation does not depend on the local state. Similarly for the callee, $\omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} I_q$. The state of other objects is not changed in the last computation step, and we get the required property.

Case: CALL_{start}, $\text{CALL}_{start}^{skip}$
These cases are analogous to the above one, where we additionally need $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \neg z'.\mathsf{started}$ and $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{started}$, respectively, to be able to apply the cooperation test. The above properties result from the antecedents $\neg started(\grave{T}, \beta)$ and $started(\grave{T}, \beta)$ of the transition, respectively, using Lemma 6.1.4 and $\grave{\omega}(z') = \beta$.

Case: $\text{CALL}_{monitor}$
As above, where $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ is implied by the transition antecedent $owns(\xi \downarrow prog, \beta)$ for the executing thread ξ, and Lemma 6.1.2.

Case: RETURN
This case is analogous to the CALL *case, where we define q_1 as the precondition of the corresponding return statement instead of the callee class invariant. The requirement $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$ of the cooperation test results from the fact that formal parameters must not be assigned to, and that method invocation statements must not contain instance variables, so that the values of the formal parameters and the expressions in the method invocation statement are untouched during the execution of the invoked method.*

For the application of the interference freedom test, to show the validity of the interferes *predicate, we use the fact that the assertion $pre(stm_3)$ neither describes the caller nor the callee, since the corresponding local configuration is not involved in the execution.*

Case: RETURN_{run}
Similar to the return case.

Case: RETURN_{wait}
In this case the antecedent $\neg owns(\grave{T} \downarrow prog, \beta)$ of the transition rule together with Lemma 6.1.3 imply $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free}$. Furthermore, the executing thread is in the notified set prior to execution, and the same lemma yields that the executing thread is registered in $\grave{\sigma}(\beta)(\mathsf{notified})$, i.e., $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \mathsf{thread}' \in z'.\mathsf{notified}$.

Case: NEW
Let $(\alpha, \grave{\tau}_1, u := \mathsf{new}; \langle \vec{y}_1 := \vec{e}_1 \rangle^{new}\, stm_1) \in \grave{T}$ be the local configuration of the executing thread prior to object creation, and $(\alpha, \acute{\tau}_1, stm_1) \in \acute{T}$ after it. Object creation updates the global state to $\hat{\sigma} = \grave{\sigma}[\beta \mapsto \sigma_{inst}^{init}[\mathsf{this} \mapsto \beta]]$, where $\beta \notin dom(\grave{\sigma})$; the executing thread's local state gets updated to $\hat{\tau}_1 = \grave{\tau}_1[u \mapsto \beta]$. After observation we have $\acute{\tau}_1 = \hat{\tau}_1[\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1}]$ and for the global state $\acute{\sigma} = \hat{\sigma}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1}]$.

In the following let $p_1 = pre(u := \mathsf{new})$, $p_2 = pre(\vec{y}_1 := \vec{e}_1)$, *and* $p_3 = post(\vec{y}_1 := \vec{e}_1)$. *By induction* $\omega, \grave{\sigma} \models_{\mathcal{G}} GI$ *and* $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_1 \wedge I$, *where* I *is the class invariant of the creator. Using the lifting lemma we get* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI \wedge P_1(z) \wedge I(z)$ *for* $\grave{\omega} = \omega[z \mapsto \alpha][\vec{v}_1 \mapsto \grave{\tau}_1(\vec{v}_1)]$ *and* $\vec{v}_1$ *the variables from the domain of* $\grave{\tau}_1$. *With Lemma 2.4.18* $\grave{\omega}[z' \mapsto dom(\grave{\sigma})][u \mapsto \beta], \hat{\sigma} \models_{\mathcal{G}} (GI \wedge (\exists u.\ P_1(z)) \wedge I(z)) \downarrow z'$. *Note that* GI *may not contain free logical variables, and thus its evaluation does not depend on the logical environment. Since the newly created object with a fresh identity is in its initial instance state,* $\grave{\omega}[z' \mapsto dom(\grave{\sigma})][u \mapsto \beta], \hat{\sigma} \models_{\mathcal{G}} \mathsf{Fresh}(z', u)$. *Thus the cooperation test for object creation implies*

$$\grave{\omega}[u \mapsto \beta], \hat{\sigma} \quad \models_{\mathcal{G}} \quad I_{\mathsf{new}}(u) \wedge (GI \wedge P_3(z))[\vec{E}_1(z)/z.\vec{y}_1]\ ,$$

where I_{new} *is the class invariant of the new object. Using the lifting lemma again but in the reverse direction and Lemma 5.1.2 results* $\omega, \acute{\sigma} \models_{\mathcal{G}} GI$, *and thus part (2). Note that in the annotation no free logical variables occur, and thus the values of assertions do not depend on the logical environment.*

Furthermore, using the substitution lemmas we get

$$\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3 \quad \textit{and} \quad \omega, \acute{\sigma}(\beta), \tau \models_{\mathcal{L}} I_{\mathsf{new}}$$

for all τ. *For the class invariant of the executing thread, the interference freedom test implies* $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} I$, *where* I *is the class invariant of* α. *Since* I *may contain instance variables only, its evaluation does not depend on the local state, and the required property holds. The states of other objects different from both* α *and* β *are not changed in the last computation step, and part (3) is satisfied.*

Furthermore, part (1) is satisfied for the local configuration involved in the last computation step. All other configurations $(\gamma, \grave{\tau}_2, stm_2)$ *in* $\acute{T}$ *are also in* $\grave{T}$ *and* $\gamma \neq \beta$. *If* $\gamma \neq \alpha$, *then* $\grave{\sigma}(\gamma) = \acute{\sigma}(\gamma)$, *and thus* $\omega, \acute{\sigma}(\gamma), \grave{\tau}_2 \models_{\mathcal{L}} pre(stm_2)$ *by induction.*

Assume now $\gamma = \alpha$, *and let* τ *be* $\hat{\tau}_1[\vec{v}' \mapsto \grave{\tau}_2(\vec{v})]$, *where* $\vec{v} = dom(\grave{\tau}_2)$. *Since* $\grave{\sigma}(\alpha) = \hat{\sigma}(\alpha)$, *Lemma 6.1.2, the induction assumptions, and the definition of* interferes *imply with the interference freedom test* $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_2)[\vec{e}_1/\vec{y}_1]$. *The substitution Lemma 5.1.1 and the fact that, due to the renaming mechanism, no local variables in* $\vec{v}'$ *occur in* $\vec{y}_1$, *yields* $\omega, \acute{\sigma}(\alpha), \grave{\tau}_2 \models_{\mathcal{L}} pre(stm_2)$. *I.e., part (1) is satisfied.* □

Proof A.2.6 (of the soundness Corollary 6.1.6) *The proof is straightforward using the soundness Theorem 6.1.5.* □

A.3 Completeness

The following lemma states that the variable loc indeed stores the current control point of a thread:

Lemma A.3.1 *Let $\langle T, \sigma\rangle$ be a reachable configuration of prog′ and assume $(\alpha, \tau, stm) \in T$. Then $\tau(\mathsf{loc}) \equiv stm$.*

Proof A.3.2 (of Lemma A.3.1) *Straightforward by the definition of augmentation.* □

Proof A.3.3 (of the local merging Lemma 6.2.3) *Assume two computations $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \acute{T}_2, \acute{\sigma}_2\rangle$ of prog′, and let $(\alpha, \tau, stm) \in \acute{T}_1$ with $\alpha \in dom(\acute{\sigma}_1) \cap dom(\acute{\sigma}_2)$ and $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{inst})$. We prove $(\alpha, \tau, stm) \in \acute{T}_2$ by induction over the sum of the length of the computations.*

In the initial case both $\acute{T}_1$ and $\acute{T}_2$ contain the same single initial local configuration, and thus the property holds.

For the inductive case, let $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ be the last steps of the computations. The augmentation definition implies that each computation step appends at most one element to the instance history of α. If $\grave{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_1(\alpha)(\mathsf{h}_{inst})$, then, by the definition of the augmentation, $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ does not execute in α, i.e., $(\alpha, \tau, stm) \in \grave{T}_1$, and the property follows by induction. The case for $\grave{\sigma}_2(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{inst})$ is analogous. Thus assume in the following $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \grave{\sigma}_1(\alpha)(\mathsf{h}_{inst}) \circ (\sigma^1_{inst}, \tau_1)$ and $\acute{\sigma}_2(\alpha)(\mathsf{h}_{inst}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{inst}) \circ (\sigma^2_{inst}, \tau_2)$. From $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{inst})$ we conclude that $\grave{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{inst})$ and $(\sigma^1_{inst}, \tau_1) = (\sigma^2_{inst}, \tau_2)$.

Since $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) \neq \grave{\sigma}_1(\alpha)(\mathsf{h}_{inst})$, the computation step $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ executes some statements in α. If there is only one local configuration in α that is involved in the step, then the augmentation definition and the local substitution lemma imply that its resulting local configuration in $\acute{T}_1$ is given by (α, τ_1, stm_1) with $stm_1 \equiv \tau_1(\mathsf{loc})$. From $(\sigma^1_{inst}, \tau_1) = (\sigma^2_{inst}, \tau_2)$ we conclude that the same local configuration executes in $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$. Thus, either $(\alpha, \tau, stm) \in \acute{T}_1$ is the executing configuration (α, τ_1, stm_1) and then it is also in $\acute{T}_2$, or not, and then it is in $\grave{T}_1$, by induction in $\grave{T}_2$, and since it is not involved in the execution $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$, also in $\acute{T}_2$.

If otherwise there are two local configurations in α involved in $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$, then the step executes a self-communication (call or return), and, due to the completeness augmentation definition, $(\sigma^1_{inst}, \tau_1)$ specifies the callee's instance local state. However, due to the built-in auxiliary variables, the identity of the caller local configuration is also stored in τ_1, in the formal parameter caller *of the callee. The caller configuration is in $\grave{T}_1$, and by induction in $\grave{T}_2$. Furthermore, since there are no two local configurations with the same identity in a reachable configuration, both steps execute a self-call in the same local configuration and the same instance state.*

Thus, either $(\alpha, \tau, stm) \in \acute{T}_1$ is one of the executing configurations and then it is also in $\acute{T}_2$, or not, and then it is in $\grave{T}_1$, by induction in $\grave{T}_2$, and since it is not involved in the execution, also in $\acute{T}_2$. □

Proof A.3.4 (of the global merging Lemma 6.2.4) *Assume two reachable configurations $\langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle \acute{T}_2, \acute{\sigma}_2\rangle$ and let $\alpha \in dom(\acute{\sigma}_1) \cap dom(\acute{\sigma}_2)$ satisfying $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$. We show that there exists a reachable $\langle \acute{T}, \acute{\sigma}\rangle$ with $dom(\acute{\sigma}) = dom(\acute{\sigma}_2)$, $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$, and $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2)\backslash\{\alpha\}$. We proceed by induction on the sum of the lengths of the computations leading to $\langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle \acute{T}_2, \acute{\sigma}_2\rangle$.*

In the base case we are given $\langle \acute{T}_1, \acute{\sigma}_1\rangle = \langle \acute{T}_2, \acute{\sigma}_2\rangle$ and the property trivially holds.

For the inductive step, let $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ be the last steps of the computations.

If $\alpha \notin dom(\grave{\sigma}_1)$ or $\alpha \notin dom(\grave{\sigma}_2)$, then α was created in one of the last steps, and thus $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm}) = \epsilon$. That means, no methods of α were involved yet, i.e., α is in its initial instance state $\acute{\sigma}_1(\alpha) = \acute{\sigma}_2(\alpha) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \alpha]$; in this case $\langle \acute{T}_2, \acute{\sigma}_2\rangle$ already satisfies the requirements. Assume in the following $\alpha \in dom(\grave{\sigma}_1) \cap dom(\grave{\sigma}_2)$. We distinguish whether the last computation steps update the communication history of α or not.

Case: $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$
In this case $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ does not execute any non-self communication or object creation in α. By induction there is a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma}\rangle$ leading to a configuration such that $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$ and $\grave{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2)\backslash\{\alpha\}$.

In case $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ does not execute in α at all, i.e., $\grave{\sigma}_1(\alpha) = \acute{\sigma}_1(\alpha)$, then $\langle \grave{T}, \grave{\sigma}\rangle$ already satisfies the requirements.

Otherwise, the local configurations in $\grave{T}_1$ which execute in α and which are involved in the computation step $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ are by the local merging Lemma 6.2.3 also in $\grave{T}$. Furthermore, from $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ we conclude that they do not execute any non-self communication or object creation, and thus their enabledness and effect depends only on the instance state of α. That means, the same computation as in $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ can be executed in $\langle \grave{T}, \grave{\sigma}\rangle$, leading to a reachable global configuration satisfying the requirements.

Case: $\grave{\sigma}_2(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$
In this case $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ does not execute any non-self communication or object creation involving α. By induction, there is a reachable $\langle \grave{T}, \grave{\sigma}\rangle$ with $\grave{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$ and $\grave{\sigma}(\beta) = \grave{\sigma}_2(\beta)$ for all $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$.

If $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ performs a step within α, then, according to the case assumption, it executes exclusively within α. This means, $\grave{\sigma}_2(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2)\backslash\{\alpha\}$, and $\langle \grave{T}, \grave{\sigma}\rangle$ already satisfies the required properties.

If otherwise $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ does not execute in α, then all local configurations in $\grave{T}_2$, executing in an object different from α, are also in $\grave{T}$; this follows from $\grave{\sigma}_2(\beta) = \grave{\sigma}(\beta)$ for all $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$, and with the help of the local merging Lemma 6.2.3 applied to $\langle \grave{T}, \grave{\sigma}\rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2\rangle$. The enabledness of local configurations, whose execution does not involve α, are independent of the instance state of α; furthermore, the effect of their execution neither influences the instance state of α nor depends on it. Thus in $\langle \grave{T}, \grave{\sigma}\rangle$ we can execute the *same computation steps as in* $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$*, leading to a reachable configuration with the required properties.*

Case: $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) \neq \acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ *and* $\grave{\sigma}_2(\alpha)(\mathsf{h}_{comm}) \neq \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$
In this case finally both $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ *and* $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ *execute some object creation or non-self communication in* α*. We show that in this case* $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ *implies also* $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{comm})$*, and thus by induction there is a computation leading to a configuration* $\langle \grave{T}, \grave{\sigma}\rangle$ *such that* $dom(\grave{\sigma}) = dom(\grave{\sigma}_2)$*,* $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$*, and* $\grave{\sigma}(\beta) = \grave{\sigma}_2(\beta)$ *for all other objects* $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$*.*

Furthermore, combining those local configurations involved in $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ *which execute within* α *with those in* $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ *which execute outside* α*, we can define a computation* $\langle \grave{T}, \grave{\sigma}\rangle \longrightarrow \langle \acute{T}, \acute{\sigma}\rangle$ *such that* $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$ *and* $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ *for all other objects* $\beta \in dom(\acute{\sigma}_2)\backslash\{\alpha\}$*.*

The case assumptions imply, that the last elements of the communication histories $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ *and* $\acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ *were appended in the last computation steps;* $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ *imply that the last elements are equal.*

According to the augmentation, each computation step extends the communication history of α *with at most one element. Thus we get* $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{comm})$*, and by induction there is a reachable* $\langle \grave{T}, \grave{\sigma}\rangle$ *with* $dom(\grave{\sigma}) = dom(\grave{\sigma}_2)$*,* $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$*, and* $\grave{\sigma}(\beta) = \grave{\sigma}_2(\beta)$ *for all* $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$*.*

Note that the last elements of the communication histories $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ *and* $\acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ *record the kind of execution, and so we know that both steps execute the same kind of communication in* α*. Furthermore, the last elements record also the identity of the local configuration executing in* α*, the communication partner of* α*, and the communicated values, which are consequently also equal.*

We distinguish on the kind of the computation step $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$*:*

Subcase: NEW
In this case $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) \circ (\alpha, null, (\mathsf{new}^c\gamma, thread_\alpha))$*, where* $thread_\alpha$ *is the identity of the creator thread as specified by its local variable* thread*, and* γ *is the newly created object.*

From the preliminary observations we conclude that $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ *creates the same new object* γ *being in the same initial state; furthermore, it leaves the states of all objects from* $dom(\grave{\sigma}_2)\backslash\{\alpha\}$ *untouched.*

As $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$*, the local merging Lemma 6.2.3 implies that the local configuration of the creator in* $\grave{T}_1$ *is also contained in* $\grave{T}$*. Thus, since* $\gamma \notin dom(\grave{\sigma}_2) = dom(\grave{\sigma})$*, the same computation step as in* $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ *can be executed also in* $\langle \grave{T}, \grave{\sigma}\rangle$*, leading to a reachable configuration* $\langle \acute{T}, \acute{\sigma}\rangle$ *with* $dom(\acute{\sigma}) = dom(\grave{\sigma}) \,\dot{\cup}\, \{\gamma\} = dom(\grave{\sigma}_2) \,\dot{\cup}\, \{\gamma\} = dom(\acute{\sigma}_2)$*,* $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$*, and* $\acute{\sigma}(\beta) = \grave{\sigma}(\beta) = \grave{\sigma}_2(\beta) = \acute{\sigma}_2(\beta)$ *for all* $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$*. Finally, for the newly created object we have* $\acute{\sigma}(\gamma) = \acute{\sigma}_2(\gamma) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \gamma]$*, and thus* $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ *for all* $\beta \in dom(\acute{\sigma}_2)\backslash\{\alpha\}$*.*

Subcase: CALL
Assume first that α *is the caller object and* $\beta \neq \alpha$ *the callee. According to the preliminary observations, also* $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ *executes the invocation of*

the same method of β, where α is the caller and β the callee. Furthermore, by the local merging lemma, the caller local configuration from $\grave{T}_1$ is also in $\grave{T}$, and its execution is also enabled in $\langle \grave{T}, \grave{\sigma} \rangle$. The last property holds also for synchronized and monitor methods, since the invocation of the same method of β by the same thread is enabled in $\langle \grave{T}_2, \grave{\sigma}_2 \rangle$, and $\grave{\sigma}_2(\beta) = \grave{\sigma}(\beta)$.

Thus the caller local configuration from $\grave{T}_1$ can execute the method invocation in $\langle \grave{T}, \grave{\sigma} \rangle$, leading to a reachable configuration $\langle \acute{T}, \acute{\sigma} \rangle$ with $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$. Furthermore, $\langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ execute the same callee observation in the same instance state $\grave{\sigma}_2(\beta) = \grave{\sigma}(\beta)$ and the same initial local state after the communication of the same actual parameter values, and thus $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$. The states of other objects are not touched, and thus $\langle \acute{T}, \acute{\sigma} \rangle$ satisfies the required properties.

Similarly, if the callee object is α, then the same caller local configuration as in $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ can execute in $\langle \grave{T}, \grave{\sigma} \rangle$ leading to a reachable configuration satisfying the requirements.

Subcase: RETURN
This case is analogous to the above case for CALL. *The computation $\langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$ is constructed from the execution of the local configuration in α which executes in $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$, together with the execution of the communication partner of α which executes in $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$.* □

Lemma A.3.5 (Initial correctness) *The proof outline $prog'$ satisfies the initial conditions of Definition 5.2.1.*

Proof A.3.6 (of Lemma A.3.5) *Let $\{p_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{p_3\}\ stm; \mathsf{return}$ be the main statement with local variables $\vec{v}$, and let I be the class invariant of the main class. We have to show for arbitrary $\sigma \in \Sigma$ and $\omega \in \Omega$ referring only to values existing in σ, that*

$$\begin{aligned} \omega, \sigma \quad \models_{\mathcal{G}} \quad & \mathsf{InitState}(z) \wedge (\forall z'.\ z' = \mathsf{null} \vee z = z') \rightarrow \\ & \qquad P_2(z) \circ f_{init} \wedge (GI \wedge P_3(z) \wedge I(z)) \circ f_{obs} \circ f_{init}\ , \end{aligned}$$

where z is of the type of the main class, z' of type Object, *and where $f_{init} = [z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]$ and $f_{obs} = [\vec{E}_2(z)/z.\vec{y}_2]$. We observe that*

$$\omega, \sigma \models_{\mathcal{G}} \mathsf{InitState}(z) \wedge (\forall z'.\ z' = \mathsf{null} \vee z' = z)$$

implies that σ is the initial global state prior to the execution of the callee observation at the beginning of the main statement, i.e., defining exactly one existing object $\omega(z) = \alpha$ being in its initial instance state $\sigma(\alpha) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \alpha]$. We start transforming the right-hand side using the substitution Lemmas 5.1.2 and

2.3.1:

$$\begin{array}{rl} & [\![P_2(z)[z,(\mathsf{null},0,\mathsf{null})/\mathsf{thread},\mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]]\!]_{\mathcal{G}}^{\omega,\sigma} \\ = & [\![P_2(z)[z,(\mathsf{null},0,\mathsf{null})/\mathsf{thread},\mathsf{caller}]]\!]_{\mathcal{G}}^{\omega[\vec{v}\mapsto\mathsf{Init}(\vec{v})],\sigma} \\ = & [\![P_2(z)]\!]_{\mathcal{G}}^{\omega[\vec{v}\mapsto\mathsf{Init}(\vec{v})][\mathsf{thread}\mapsto\alpha],\sigma} \\ = & [\![p_2]\!]_{\mathcal{L}}^{\omega,\sigma(\alpha),\tau} \end{array}$$

with τ *defined by* $\tau_{init}[\mathsf{thread}\mapsto\alpha]$. *Note that the initial value* $\mathsf{Init}(\mathsf{caller})$ *of the variable* caller *is* $(null, 0, null)$. *The above value* $[\![p_2]\!]_{\mathcal{L}}^{\omega,\sigma(\alpha),\tau}$ *is true due to the completeness annotation definition, since the* run *method of the main class is initially invoked in the given context.*

For the global invariant we get similarly

$$\begin{array}{rl} & [\![GI[\vec{E}_2(z)/z.\vec{y}_2][z,(\mathsf{null},0,\mathsf{null})/\mathsf{thread},\mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]]\!]_{\mathcal{G}}^{\omega,\sigma} \\ = & [\![GI[\vec{E}_2(z)/z.\vec{y}_2]]\!]_{\mathcal{G}}^{\omega[\vec{v}\mapsto\mathsf{Init}(\vec{v})][\mathsf{thread}\mapsto\alpha],\sigma} \\ = & [\![GI]\!]_{\mathcal{G}}^{\omega',\sigma'} \\ = & [\![GI]\!]_{\mathcal{G}}^{\omega,\sigma'} \end{array}$$

for some logical environment ω' *and for* σ' *given by* $\sigma[\alpha.\vec{y}_2\mapsto[\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]$. *In the last step we used the restriction that the global invariant may not contain free logical variables. The step before made use of the following equation for* $\vec{E}_2(z)$, *which we get using Lemma 2.3.1 and with the fact that* $\vec{e}_2$ *does not contain logical variables:*

$$\begin{array}{rcl} [\![\vec{E}_2(z)]\!]_{\mathcal{G}}^{\omega[\vec{v}\mapsto\mathsf{Init}(\vec{v})][\mathsf{thread}\mapsto\alpha],\sigma} & = & [\![\vec{e}_2[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega[\vec{v}\mapsto\mathsf{Init}(\vec{v})][\mathsf{thread}\mapsto\alpha],\sigma} \\ & = & [\![\vec{e}_2]\!]_{\mathcal{L}}^{\omega[\vec{v}\mapsto\mathsf{Init}(\vec{v})][\mathsf{thread}\mapsto\alpha],\sigma(\alpha),\tau} \\ & = & [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} . \end{array}$$

Since $\langle T',\sigma'\rangle$ *with* $T' = \{(\alpha,\tau',stm)\}$ *and* $\tau' = \tau[\vec{y}_2\mapsto[\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]$ *is an initial global configuration of* $prog'$ *after the observation at the beginning of the main statement, it is reachable, and the initial condition for the global invariant is satisfied. The cases for* p_3 *and* I *are similar to that of* GI, *where we additionally use the lifting substitution Lemma 2.3.1 to show that* $[\![P_3(z)]\!]_{\mathcal{G}}^{\omega',\sigma'} = [\![p_3]\!]_{\mathcal{L}}^{\omega',\sigma'(\alpha),\tau'}$.
□

Lemma A.3.7 (Local correctness) *The proof outline* $prog'$ *satisfies the conditions of local correctness from Definition 5.2.2.*

Proof A.3.8 (of Lemma A.3.7) *Let* c *be a class of* $prog'$ *with class invariant* I, $\omega\in\Omega$, $\sigma_{inst}\in\Sigma_{inst}$, *and* $\tau\in\Sigma_{loc}$ *with* $\sigma_{inst}(\mathsf{this}) = \alpha$. *Assume a multiple* *assignment* $\{p_1\}\,\vec{y} := \vec{e}\{p_2\}$ *in* c *which is not the observation of communication or object creation. We have to show that*

$$\omega, \sigma_{inst}, \tau \quad \models_{\mathcal{L}} \quad p_1 \wedge I \rightarrow p_2[\vec{e}/\vec{y}] \ .$$

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p_1$ *it follows by the definition of the annotation that there is a reachable* $\langle \grave{T}, \grave{\sigma} \rangle$ *with* $\grave{\sigma}(\alpha) = \sigma_{inst}$ *and* $(\alpha, \tau, \vec{y} := \vec{e}; stm) \in \grave{T}$. *Executing in the local configuration in* $\langle \grave{T}, \grave{\sigma} \rangle$ *leads to a reachable global configuration* $\langle \acute{T}, \acute{\sigma} \rangle$ *with* $\acute{\sigma}(\alpha) = \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}]$ *and* $(\alpha, \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}], stm) \in \acute{T}$. *Thus by the definition of the annotation for prog′ we have*

$$\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}] \models_{\mathcal{L}} p_2 \ ,$$

and further with the substitution Lemma 5.1.1 $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p_2[\vec{e}/\vec{y}]$, *as required.* □

Lemma A.3.9 (Interference freedom) *The proof outline prog′ satisfies the conditions for interference freedom from Definition 5.2.3.*

Proof A.3.10 (of Lemma A.3.9) *Assume an arbitrary assignment* $\vec{y} := \vec{e}$ *with precondition* p *in class* c *with class invariant* I, *and an arbitrary assertion* q *at a control point in the same class. We show the verification condition from Equation (5.4) on page 77*

$$\omega, \sigma_{inst}, \tau \quad \models_{\mathcal{L}} \quad p \wedge q' \wedge I \wedge \mathsf{interferes}(q, \vec{y} := \vec{e}) \rightarrow q'[\vec{e}/\vec{y}] \ ,$$

for some logical environment ω *together with some instance and local states* σ_{inst} *and* τ, *where* q' *denotes* q *with all local variables* u *replaced by some fresh local variables* u'.

Let $\alpha = \sigma_{inst}(\mathsf{this})$, *and assume first that* $\vec{y} := \vec{e}$ *is not the observation of communication or object creation. The first clause* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ *implies that there exists a computation reaching* $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ *with* $\grave{\sigma}_p(\alpha) = \sigma_{inst}$, *and a configuration* $(\alpha, \tau, \vec{y} := \vec{e}; stm'_p) \in \grave{T}_p$.

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'$ *we get by renaming back the local variables that* $\omega, \sigma_{inst}, \tau' \models_{\mathcal{L}} q$ *for* $\tau'(u) = \tau(u')$ *for all local variables* u *in* q. *Let* q *be the precondition of the statement* stm_q. *Note that* q *is an assertion at a control point. Applying the annotation definition we conclude that there is a reachable* $\langle \grave{T}_q, \grave{\sigma}_q \rangle$ *with* $\grave{\sigma}_q(\alpha) = \sigma_{inst} = \grave{\sigma}_p(\alpha)$ *and* $(\alpha, \tau', stm_q; stm'_q) \in \grave{T}_q$. *The local merging Lemma 6.2.3 implies that* $(\alpha, \tau', stm_q; stm'_q) \in \grave{T}_p$.

Let $\langle \acute{T}_p, \acute{\sigma}_p \rangle$ *result from* $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ *by executing in the enabled local configuration* $(\alpha, \tau, \vec{y} := \vec{e}; stm'_p)$. *We have* $\acute{\sigma}_p(\alpha) = \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}]$. *From the assumption* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{interferes}(q, \vec{y} := \vec{e})$ *we get that* $(\alpha, \tau', stm_q; stm'_q)$ *is not the executing configuration, and thus* $(\alpha, \tau', stm_q; stm'_q) \in \acute{T}_p$.

According to the annotation definition $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}], \tau' \models_{\mathcal{L}} q$, *and after renaming the local variables of* q *also* $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}], \tau \models_{\mathcal{L}} q'$. *Due to renaming, no local variables of* q' *occur in* $\vec{y}$, *implying*

$$\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}] \models_{\mathcal{L}} q' \ .$$

Finally, by the substitution Lemma 5.1.1 we get $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$.

If the assignment observes object creation or communication, the proof is similar. For object creation, $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ *implies that there exists a computation reaching* $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ *with* $\grave{\sigma}_p(\alpha) = \sigma_{inst}$*, and an enabled configuration* $(\alpha, \tau_p, stm_p; stm'_p) \in \grave{T}_p$*, where* stm_p *is of the form* $u := \mathsf{new}; \langle \vec{y} := \vec{e} \rangle^{new}$. *The local state* τ_p *is* $\tau[u \mapsto v]$ *for some value* v*, such that the local configuration is enabled to create* $\tau(u)$*. Directly after creation, the creator local configuration has the local state* τ *and executes its observation resulting in the local state* $\tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}]$ *and instance state* $\sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}]$*. Note that* σ_{inst} *is not influenced by the object creation itself. Again, the* interferes *predicate assures that* $(\alpha, \tau', stm_q; stm'_q)$ *is not the executing configuration, and we get* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$ *as above.*

The case for caller observation in a non-self communication is analogous. In the case of caller observation in a self-communication, the restrictions on the augmentation imply that $\vec{y} := \vec{e}$ *does not change the values of instance variables, and the requirement follows directly from the assumptions. If* p *is the precondition of a callee observation at the beginning of a method body, then the annotation assures that the invocation of the method is enabled in* $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ *such that* τ *is the local state of the callee directly after communication but before observation. Note that for self-communication, the caller part does not change the instance state. Thus the only update of the instance state of* α *is given by the effect of* $\vec{y} := \vec{e}$*. Again, the* interferes *predicate assures that* $(\alpha, \tau', stm_q; stm'_q)$ *is neither the caller nor the callee, and thus* $(\alpha, \tau', stm_q; stm'_q) \in \acute{T}_p$*. We get* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$ *as above.*

Validity of the verification condition 5.3 for the class invariant is similar, where we additionally use the fact that the class invariant refers to instance variables only. □

Lemma A.3.11 (Cooperation test: Communication) *The proof outline* $prog'$ *satisfies the verification conditions of the cooperation test for communication of Definition 5.2.4.*

Proof A.3.12 (of Lemma A.3.11) *We distinguish on the kind of communication starting with the verification condition for synchronized method invocation.*

Case: CALL

Let $\{p_1\}\, u_{ret} := e_0.m(\vec{e}); \{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_3\}^{wait}$ *be a statement in a class* c *of* $prog'$ *with* e_0 *of type* c'*, where method* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *of* c' *is synchronized with body* $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\}\, stm$*, formal parameters* $\vec{u}$*, local variables without the formal parameters given by* $\vec{v}$*, and let* $q_1 = I_{c'}$ *be the callee class invariant. Assume*

$$\grave{\omega}, \grave{\sigma} \quad \models_{\mathcal{G}} \quad GI \wedge P_1(z) \wedge Q'_1(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}$$

for distinct and fresh $z \in LVar^c$ *and* $z' \in LVar^{c'}$*, and where* comm *is* $E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$*. Note that for completeness we do not need the information stored in the caller class invariant. By definition of the global invariant, the assumption* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI$ *implies that there exists a reachable* $\langle T, \sigma \rangle$ *with*

$$dom(\grave{\sigma}) = dom(\sigma) \text{ and } \grave{\sigma}(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm}) \text{ for all } \gamma \in dom(\sigma)\,.$$

Assuming $\grave{\omega}(z) = \alpha$ *as caller identity,* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} P_1(z)$ *implies* $\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_1$ *by the substitution Lemma 2.3.1, for some local state* $\grave{\tau}_1$ *with* $\grave{\tau}_1(u) = \grave{\omega}(u)$ *for all local variables* u *occurring in* p_1*. By the annotation definition there exists a reachable configuration* $\langle T_1, \sigma_1 \rangle$ *such that*

$$\sigma_1(\alpha) = \grave{\sigma}(\alpha) \text{ and } (\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call}\, stm_1) \in T_1\,.$$

Recall that $\sigma(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ *for all* $\gamma \in dom(\sigma)$*, and especially for the caller* $\sigma(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}(\alpha)(\mathsf{h}_{comm}) = \sigma_1(\alpha)(\mathsf{h}_{comm})$*. Using the global merging Lemma 6.2.4 applied to* $\langle T_1, \sigma_1 \rangle$ *and* $\langle T, \sigma \rangle$ *we get that there is a reachable* $\langle T', \sigma' \rangle$ *with* $dom(\sigma') = dom(\sigma)$ *and*

$$\sigma'(\alpha) = \sigma_1(\alpha) \text{ and } \sigma'(\gamma) = \sigma(\gamma) \text{ for all } \gamma \in dom(\sigma) \backslash \{\alpha\}\,.$$

Furthermore, $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call}\, stm_1) \in T_1$*,* $\sigma_1(\alpha) = \sigma'(\alpha)$*, and the local merging Lemma 6.2.3 implies that*

$$(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call}\, stm_1) \in T'\,.$$

Let $\beta = \grave{\omega}(z')$ *be the callee object. In case of a self-call, i.e., for* $\alpha = \beta$*, we directly get that* $\langle T'', \sigma'' \rangle = \langle T', \sigma' \rangle$ *is a reachable configuration such that* $\sigma''(\alpha) = \grave{\sigma}(\alpha)$*,* $\sigma''(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ *for all* $\gamma \in dom(\grave{\sigma})$*, and* $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call}\, stm_1) \in T''$*.*

Otherwise, the assumption $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} I_{c'}(z')$ *implies* $\grave{\omega}, \grave{\sigma}(\beta), \tau_2 \models_{\mathcal{L}} I_{c'}$ *for some local state* τ_2*. Note that the class invariant contains instance variables, only. By definition of the class invariant, there is a reachable global configuration* $\langle T_2, \sigma_2 \rangle$ *such that*

$$\sigma_2(\beta) = \grave{\sigma}(\beta)\,.$$

We need to fall back upon the two merging lemmas once more to obtain a common reachable configuration: Analogously to the caller part, the global merging Lemma 6.2.4 applied to $\langle T_2, \sigma_2 \rangle$ *and* $\langle T', \sigma' \rangle$ *yields that there is a reachable configuration* $\langle T'', \sigma'' \rangle$ *with* $dom(\sigma'') = dom(\sigma')$ *and*

$$\sigma''(\beta) = \sigma_2(\beta) \text{ and } \sigma''(\gamma) = \sigma'(\gamma) \text{ for all } \gamma \in dom(\sigma') \backslash \{\beta\}\,.$$

Now, $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call}\, stm_1) \in T'$*,* $\sigma''(\alpha) = \sigma'(\alpha)$*, and the local merging Lemma 6.2.3 implies that the local configuration* $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call}\, stm_1)$ *is in* T''*.*

Thus $\langle T'', \sigma'' \rangle$ *is a reachable configuration with* $\sigma''(\alpha) = \grave{\sigma}(\alpha)$*,* $\sigma''(\beta) = \grave{\sigma}(\beta)$*,* $\sigma''(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ *for all* $\gamma \in dom(\grave{\sigma})$*, and* $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call}\, stm_1) \in T''$*.*

With the antecedent $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *of the cooperation test we get* $\grave{\sigma}(\beta)(\mathsf{lock}) = \mathit{free} \vee \mathit{thread}(\grave{\sigma}(\beta)(\mathsf{lock})) = \grave{\tau}_1(\mathsf{thread})$. *With* $\grave{\sigma}(\beta) = \sigma''(\beta)$ *and Lemma 6.1.3 we get* $\neg \mathit{owns}(T''\backslash\{\xi\}, \beta)$, *where* ξ *is the stack with* $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\, \mathit{stm}_1)$ *on top. Furthermore,* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \mathsf{comm}$ *implies* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} E_0(z) = z'$, *and by the lifting substitution lemma* $[\![e_0]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1} = [\![e_0]\!]_{\mathcal{E}}^{\sigma''(\alpha), \grave{\tau}_1} = \grave{\omega}(z') = \beta$. *This means, the invocation of method* m *of* β *is enabled in the local configuration* $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\, \mathit{stm}_1)$ *in* $\langle T'', \sigma''\rangle$.

The definition of the augmentation, and $\sigma''(\alpha) = \grave{\sigma}(\alpha)$ *gives*

$$\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_2\,,$$

which by the substitution Lemma 2.3.1 and with the definition of $\grave{\tau}_1$ *yields* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} P_2(z)$. *Due to the renaming mechanism we get*

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} P_2(z) \circ f_{comm}$$

for $f_{comm} = [\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$. *For the precondition of the method body, the annotation definition implies*

$$\grave{\omega}, \grave{\sigma}(\beta), \hat{\tau}_2 \models_{\mathcal{L}} q_2$$

with $\hat{\tau}_2 = \tau_{init}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$. *For the actual parameters we obtain by the substitution Lemma 2.3.1* $[\![\vec{E}(z)]\!]_{\mathcal{G}}^{\grave{\omega}, \grave{\sigma}} = [\![\vec{e}]\!]_{\mathcal{L}}^{\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1} = [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}$, *and further with the same lemma*

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} Q_2'(z')[\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$$

as required by the cooperation test.

Directly after communication we have a global configuration with still the same global state σ''. *The caller observation evolves its own local state to* $\acute{\tau}_1 = \grave{\tau}_1[\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma''(\alpha), \grave{\tau}_1}]$, *and the global state to* $\hat{\sigma} = \sigma''[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma''(\alpha), \grave{\tau}_1}]$. *Finally, the callee observation changes the global state to* $\acute{\sigma} = \hat{\sigma}[\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$, *where its own local state is updated to* $\acute{\tau}_2 = \hat{\tau}_2[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$. *According to the annotation definition we get*

$$\grave{\omega}, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3, \quad \grave{\omega}, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} q_3, \quad \text{and} \quad \grave{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI\,.$$

Let $\acute{\omega} = \grave{\omega}[\vec{v}' \mapsto \mathit{Init}(\vec{v})][\vec{u}' \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}][\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}][\vec{y}_2' \mapsto [\![\vec{e}_2']\!]_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$. *The lifting lemma implies* $\acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI \wedge P_3(z) \wedge Q_3'(z')$; *with the global substitution lemma finally*

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z) \wedge Q_3'(z'))[\vec{E}_2'(z')/z'.\vec{y}_2'][\vec{E}_1(z)/z.\vec{y}_1][\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']\,,$$

and thus the cooperation test is satisfied for the invocation of synchronous methods.

The case for non-synchronized methods is analogous, where the antecedent $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *is dropped.*

Case: $\textsc{Call}_{monitor}$
This case is similar to the above one of $\textsc{Call}$*, where for the invocation of a method* $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$*, the assertion* comm *is given by* $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$*, implying* $owns(\xi, \beta)$ *for the caller thread* ξ *and the callee object* β*.*

Case: $\textsc{Call}_{start}$
Enabledness of starting the thread of an object β *requires* $\neg started(T'', \beta)$*. Due to the definition of* comm*, we have additionally* $\grave{\omega}, \sigma'' \models_{\mathcal{G}} \neg z'.\mathsf{started}$*, which implies* $\neg\sigma''(\beta)(\mathsf{started})$*. We get enabledness by Lemma 6.1.4.*

Case: $\textsc{Call}_{start}^{skip}$
The enabledness argument is similar for $\textsc{Call}_{start}^{skip}$*, where we use* $\grave{\omega}, \sigma'' \models_{\mathcal{G}} z'.\mathsf{started}$ *to imply the enabledness predicate* $started(T'', \beta)$*.*

Case: $\textsc{Return}$
For return, the construction of $\langle T'', \sigma'' \rangle$ *is similar, where we get instead of the enabledness of the caller that the callee configuration* $(\beta, \grave{\tau}_2, \mathsf{return}\ e_{ret}; \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret})$ *is in* $\langle T'', \sigma'' \rangle$*, and thus enabled to execute.*

Case: $\textsc{Return}_{wait}$
In this case we additionally have to show $\neg owns(T'', \beta)$*, which we get from the* comm *assertion implying* $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free}$ *and using Lemma 6.1.3.*

Case: $\textsc{Return}_{run}$
Since the run *method cannot be invoked directly, we conclude that the executing local configuration is the only one in its stack, i.e., the transition rule* $\textsc{Return}_{run}$ *of the semantics can be applied in* $\langle T'', \sigma'' \rangle$ *to terminate the callee* $(\beta, \grave{\tau}_2, \mathsf{return}; \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret})$*.*

$\square$

Lemma A.3.13 (Cooperation test: Instantiation) *The proof outline* $prog'$ *satisfies the verification conditions of the cooperation test for object creation of Definition 5.2.5.*

Proof A.3.14 (of Lemma A.3.13) *Let* $\{p_1\}\, u := \mathsf{new}^c; \{p_2\}^{new}\, \langle \vec{y} := \vec{e} \rangle^{new}\, \{p_3\}$ *be a statement in class* c' *of* $prog'$*, and assume*

$$\hat{\omega}, \hat{\sigma} \quad \models_{\mathcal{G}} \quad z \neq \mathsf{null} \wedge z \neq u \wedge \exists z'.\ \mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u(P_1(z))) \downarrow z'$$

with $z \in LVar^{c'}$ *and* $z' \in LVar^{\mathsf{list\ Object}}$ *fresh. Note that we do not need the class invariant of the creator for completeness. We show that*

$$\hat{\omega}, \hat{\sigma} \quad \models_{\mathcal{G}} \quad P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z))[\vec{E}(z)/z.\vec{y}]\ .$$

Let $\hat{\omega}(z) = \alpha$ *and* $\hat{\omega}(u) = \beta$*. According to the semantics of assertions we have that*

$$\omega, \hat{\sigma} \quad \models_{\mathcal{G}} \quad \mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'$$

for some logical environment ω *that assigns to* z' *a sequence of objects from* $Val^{\mathsf{Object}}_{null}(\hat{\sigma}) = \bigcup_c Val^c_{null}(\hat{\sigma})$*, and agrees on the values of all other variables with* $\hat{\omega}$*. The assertion* $\mathsf{Fresh}(z', u)$ *is defined by*

$$\mathsf{InitState}(u) \wedge u \notin z' \wedge \forall v.\ v \in z' \vee v = u\ ,$$

where $\mathsf{InitState}(u)$ *expands to* $u \neq \mathsf{null} \wedge \bigwedge_{x \in IVar_c} u.x = \mathsf{Init}(x)$*. Thus,* $\omega, \hat{\sigma} \models_{\mathcal{G}} \mathsf{Fresh}(z', u)$ *implies that* $\beta \in Val^c(\hat{\sigma})$ *with* $\hat{\sigma}(\beta) = \sigma^{init}_{inst}[\mathsf{this} \mapsto \beta]$*, and additionally* $Val^{\mathsf{Object}}_{null}(\hat{\sigma}) = \omega(z')\ \dot{\cup}\ \{\beta\}$*. Let* $\grave{\sigma}$ *be the global state with domain* $Val^{\mathsf{Object}}(\grave{\sigma}) = Val^{\mathsf{Object}}(\hat{\sigma}) \backslash \{\beta\}$ *and such that* $\grave{\sigma}(\gamma) = \hat{\sigma}(\gamma)$ *for all objects* $\gamma \in Val^{\mathsf{Object}}(\grave{\sigma})$*. Then* $\hat{\sigma} = \grave{\sigma}[\beta \mapsto \sigma^{init}_{inst}[\mathsf{this} \mapsto \beta]]$*, and from*

$$\omega, \hat{\sigma} \models_{\mathcal{G}} (GI \wedge \exists u.\ P_1(z))\ \downarrow z'$$

we get with Lemma 2.4.18

$$\omega, \grave{\sigma} \models_{\mathcal{G}} GI \wedge \exists u.\ P_1(z)\ .$$

By definition of the annotation, $\omega, \grave{\sigma} \models_{\mathcal{G}} GI$ *implies that there is a reachable configuration* $\langle \grave{T}_1, \grave{\sigma}_1 \rangle$ *such that*

$$dom(\grave{\sigma}_1) = dom(\grave{\sigma}) \text{ and } \grave{\sigma}_1(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm}) \text{ for all } \gamma \in dom(\grave{\sigma})\,.$$

The precondition of the object creation statement

$$\omega, \grave{\sigma} \models_{\mathcal{G}} \exists u.\ P_1(z)$$

implies

$$\omega[u \mapsto v], \grave{\sigma} \models_{\mathcal{G}} P_1(z)$$

for some $v \in Val^{\mathsf{Object}}_{null}(\grave{\sigma})$*. Applying the lifting Lemma 2.3.1 we get that*

$$\omega, \grave{\sigma}(\alpha), \grave{\tau} \models_{\mathcal{L}} p_1$$

for a local state $\grave{\tau}$ *with* $\grave{\tau}(u) = v$ *and* $\grave{\tau}(w) = \omega(w)$ *for all other local variables* w*. By definition of the annotation, there is a reachable global configuration* $\langle \grave{T}_2, \grave{\sigma}_2 \rangle$ *such that*

$$\grave{\sigma}_2(\alpha) = \grave{\sigma}(\alpha) \text{ and } (\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{new}\, stm) \in \grave{T}_2\,.$$

Recall that $\grave{\sigma}_1(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ *for all* $\gamma \in dom(\grave{\sigma})$*; especially we have* $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{comm})$*. Using the global merging Lemma 6.2.4 applied to the reachable global configurations* $\langle \grave{T}_2, \grave{\sigma}_2 \rangle$ *and* $\langle \grave{T}_1, \grave{\sigma}_1 \rangle$ *we get that there is a reachable configuration* $\langle \grave{T}_3, \grave{\sigma}_3 \rangle$ *with*

$$dom(\grave{\sigma}_3) = dom(\grave{\sigma}_1),\ \grave{\sigma}_3(\alpha) = \grave{\sigma}_2(\alpha), \text{ and } \grave{\sigma}_3(\gamma) = \grave{\sigma}_1(\gamma) \text{ for all } \gamma \in dom(\grave{\sigma}_1) \backslash \{\alpha\}.$$

Furthermore, $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{new}\, stm) \in \grave{T}_2$*,* $\grave{\sigma}_2(\alpha) = \grave{\sigma}_3(\alpha)$*, and the local merging Lemma 6.2.3 implies that* $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{new}\, stm) \in \grave{T}_3$.

So we know that $\langle\grave{T}_3,\grave{\sigma}_3\rangle$ *is a reachable configuration containing the local configuration* $(\alpha,\grave{\tau},u:=\mathsf{new}^c;\langle\vec{y}:=\vec{e}\rangle^{new}\,stm)\in\grave{T}_3$. *With* $Val^{\mathsf{Object}}(\grave{\sigma})=Val^{\mathsf{Object}}(\hat{\sigma})\backslash\{\beta\}$, $dom(\grave{\sigma}_1)=dom(\grave{\sigma})$, *and* $dom(\grave{\sigma}_3)=dom(\grave{\sigma}_1)$ *we get that* $\beta\notin dom(\grave{\sigma}_3)$, *i.e., the local configuration is enabled to create the fresh object* $\beta=\omega(u)$. *With* $\grave{\sigma}_3(\alpha)=\grave{\sigma}_2(\alpha)=\hat{\sigma}(\alpha)$ *we get*

$$\omega,\hat{\sigma}(\alpha),\hat{\tau}\models_{\mathcal{L}} p_2\ ,$$

where $\hat{\tau}=\grave{\tau}[u\mapsto\beta]$*; with the lifting Lemma 2.3.1 together with the definition of* $\grave{\tau}$ *this means* $\omega,\hat{\sigma}\models_{\mathcal{G}} P_2(z)$, *as required in the cooperation test.*

Executing the instantiation in the local configuration $(\alpha,\grave{\tau},u:=\mathsf{new}^c;\langle\vec{y}:=\vec{e}\rangle^{new}\,stm)$ *in* $\langle\grave{T}_3,\grave{\sigma}_3\rangle$, *creating a new object* $\beta\notin dom(\grave{\sigma}_3)$, *results in* $\langle\hat{T}_3,\hat{\sigma}_3\rangle$ *with* $\hat{\sigma}_3=\grave{\sigma}_3[\beta\mapsto\sigma^{init}_{inst}[\mathsf{this}\mapsto\beta]]$*; executing the creator observation leads to a reachable* $\langle\acute{T}_3,\acute{\sigma}_3\rangle$ *with* $\acute{\sigma}_3=\hat{\sigma}_3[\alpha.\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}_3(\alpha),\hat{\tau}}]$ *and* $(\alpha,\acute{\tau},stm)$ *in* $\acute{T}_3$ *with* $\acute{\tau}=\hat{\tau}[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}_3(\alpha),\hat{\tau}}]$.

As $\langle\acute{T}_3,\acute{\sigma}_3\rangle$ *is reachable with* $\acute{\sigma}_3(\beta)=\sigma^{init}_{inst}[\mathsf{this}\mapsto\beta]=\hat{\sigma}(\beta)$ *we know*

$$\hat{\omega},\hat{\sigma}(\beta),\acute{\tau}\models_{\mathcal{L}} I_c\,.$$

As I_c *may not contain local variables, applying the lifting Lemma 2.3.1 again with* $\omega(u)=\beta$ *yields the required condition* $\hat{\omega},\hat{\sigma}\models_{\mathcal{G}} I_c(u)$ *for the class invariant. It remains to show that*

$$\hat{\omega},\hat{\sigma}\quad\models_{\mathcal{G}}\quad(GI\wedge P_3(z))[\vec{E}(z)/z.\vec{y}]\ .$$

Applying the substitution Lemma 5.1.2 and the fact that GI *does not contain free logical variables yields*

$$[\![GI[\vec{E}(z)/z.\vec{y}]]\!]_{\mathcal{G}}^{\hat{\omega},\hat{\sigma}}\quad=\quad[\![GI]\!]_{\mathcal{G}}^{\hat{\omega},\acute{\sigma}}$$

with $\acute{\sigma}=\hat{\sigma}[\alpha.\vec{y}\mapsto[\![\vec{E}(z)]\!]_{\mathcal{G}}^{\hat{\omega},\hat{\sigma}}]$. *Thus we have to show the existence of a reachable configuration with a global state defining the same object domain and communication history values as* $\acute{\sigma}$. *The configuration* $\langle\acute{T}_3,\acute{\sigma}_3\rangle$ *satisfies the above requirements, since, first, it is reachable with*

$$\begin{aligned} dom(\acute{\sigma}_3) &= dom(\grave{\sigma}_3)\,\dot{\cup}\,\{\beta\}=dom(\grave{\sigma}_1)\,\dot{\cup}\,\{\beta\}\\ &= dom(\grave{\sigma})\,\dot{\cup}\,\{\beta\}=dom(\hat{\sigma})=dom(\acute{\sigma})\,. \end{aligned}$$

Furthermore, $\acute{\sigma}_3(\alpha)=\hat{\sigma}_3(\alpha)[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}_3(\alpha),\hat{\tau}}]$, *and with* $\hat{\sigma}_3(\alpha)=\grave{\sigma}_3(\alpha)=\grave{\sigma}_2(\alpha)=\hat{\sigma}(\alpha)$ *and*

$$[\![\vec{E}(z)]\!]_{\mathcal{G}}^{\hat{\omega},\hat{\sigma}}=[\![\vec{e}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\hat{\omega},\hat{\sigma}}=[\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}(\alpha),\hat{\tau}}=[\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}_3(\alpha),\hat{\tau}}\ ,$$

we get $\acute{\sigma}_3(\alpha)=\acute{\sigma}(\alpha)$. *For the new object,* $\acute{\sigma}_3(\beta)=\hat{\sigma}_3(\beta)=\sigma^{init}_{inst}[\mathsf{this}\mapsto\beta]=\hat{\sigma}(\beta)=\acute{\sigma}(\beta)$. *Finally, for all other objects* γ *different from both* α *and* β *from the domain of* $\acute{\sigma}$ *we have* $\acute{\sigma}_3(\gamma)(\mathsf{h}_{comm})=\grave{\sigma}_3(\gamma)(\mathsf{h}_{comm})=\grave{\sigma}_1(\gamma)(\mathsf{h}_{comm})=\acute{\sigma}(\gamma)(\mathsf{h}_{comm})$.

Similarly for the postcondition p_3 of the observation,

$$\begin{aligned}[\![P_3(z)[\vec{E}(z)/z.\vec{y}]]\!]_{\mathcal{G}}^{\hat{\omega},\hat{\sigma}} &= [\![P_3(z)]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} \\ &= [\![p_3[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} = [\![p_3]\!]_{\mathcal{L}}^{\acute{\omega},\acute{\sigma}(\alpha),\acute{\tau}} = [\![p_3]\!]_{\mathcal{L}}^{\acute{\omega},\acute{\sigma}_3(\alpha),\acute{\tau}} .\end{aligned}$$

Thus we have to show the existence of a reachable configuration with a global state defining the same instance state for α as $\acute{\sigma}_3$ and containing the local configuration $(\alpha, \acute{\tau}, stm)$. The configuration $\langle \acute{T}_3, \acute{\sigma}_3 \rangle$ satisfies the above requirements. □

Proof A.3.15 (of Theorem 6.2.5) *Straightforward using the Lemmas A.3.5, A.3.7, A.3.9, A.3.11, and A.3.13.* □

Appendix B

Deadlock freedom examples

B.1 Reentrant monitors

$$
\begin{array}{l}
GI \stackrel{def}{=} \\
\quad (\forall (z : Synch).z \neq null \rightarrow \\
\qquad (z.lock = (null, 0) \vee \\
\qquad (\exists (t : Main).owns(t, z.lock) \wedge t.started \wedge t.created = z) \vee \\
\qquad (owns(z, z.lock) \wedge z.started))) \wedge \\
\quad (\forall (t : Main).(t \neq null \wedge \neg t.in\_Synch) \rightarrow (t.created = null \vee not\_owns(t, t.created.lock))) \wedge \\
\quad (\forall (t : Main).t \neq null \rightarrow (\forall (z : Synch).(z \neq null \wedge owns(t, z.lock)) \rightarrow t.created = z))
\end{array}
$$

$I_{Main} \stackrel{def}{=} started$

```
class Main{
    ⟨ Bool in_Synch; ⟩
    ⟨ Synch created; ⟩

    nsync Void wait(){ {false}^?call {false} return_getlock {false}^!ret }

    nsync Void run(){
        Synch obj;
        {thread = this ∧ ¬in_Synch ∧ created = null ∧ conf = 0}
        obj := new^Synch; {thread = this ∧ conf = 0}^new ⟨created := obj⟩^new
        {obj ≠ null ∧ obj ≠ this ∧ thread = this ∧ ¬in_Synch ∧ created = obj ∧ conf = 0}
        obj.start();
        {obj ≠ null ∧ obj ≠ this ∧ thread = this ∧ ¬in_Synch ∧ created = obj ∧ conf = 0}
        obj.m1()
        {thread = this ∧ conf = 0}^!call
            ⟨in_Synch := (if obj = this then in_Synch else true fi)⟩^!call
        {thread = this ∧ created = obj ∧ conf = 0}^wait
        {thread = this ∧ conf = 0}^?ret
            ⟨in_Synch := (if obj = this then in_Synch else false fi)⟩^?ret
        {thread = this ∧ ¬in_Synch ∧ created = obj ∧ conf = 0}
    }
}

class Synch{

    nsync Void wait(){ {false}^?call {false} return_getlock {false}^!ret }

    sync Void m1(){
        {owns(thread, lock) ∧ depth(lock) = 1}
```

```
        m2()
        {owns(thread, lock) ∧ depth(lock) = 1}
    }

    sync Void m2(){
        {owns(thread, lock) ∧ depth(lock) = 2}
    }

    nsync Void run(){
        {thread = this ∧ started ∧ not_owns(thread, lock)}
        m1()
        {not_owns(thread, lock)}
    }

}
```

B.2 A simple wait-notify example

$$
\begin{aligned}
&GI \stackrel{def}{=}\\
&\quad (\forall(z_1, z_2 : Main).(z_1 \neq null \wedge z_2 \neq null) \rightarrow z_1 = z_2) \wedge\\
&\quad (\forall(z_1, z_2 : Monitor).(z_1 \neq null \wedge z_2 \neq null) \rightarrow z_1 = z_2) \wedge\\
&\quad (\forall(z : Main).z \neq null \rightarrow\\
&\qquad (z.started \wedge z.x \geq 0 \wedge z.x \leq 3 \wedge\\
&\qquad (z.x = 0 \rightarrow z.created = null \wedge (\forall(z_2 : Monitor).z_2 = null)) \wedge\\
&\qquad (z.x = 1 \rightarrow (z.created \neq null \wedge z.created \neq z \wedge z.created.lock = (null, 0) \wedge\\
&\qquad\quad z.created.x = 0 \wedge length(z.created.wait) = 0 \wedge length(z.created.notified) = 0 \wedge\\
&\qquad\quad z.created.counter = 0 \wedge \neg z.created.started)) \wedge\\
&\qquad (z.x = 3 \rightarrow z.created \neq null \wedge not\_owns(z, z.created.lock) \wedge z.created.x = 8) \wedge\\
&\qquad (z.x = 2 \rightarrow z.created \neq null))) \wedge\\
&\quad (\forall(z_1 : Main).z_1 \neq null \rightarrow (\forall(z_2 : Monitor).(z_2 \neq null \wedge owns(z_1, z_2.lock)) \rightarrow\\
&\qquad z_2 = z_1.created)) \wedge\\
&\quad (\forall(z_1, z_2 : Monitor).(z_1 \neq null \wedge z_2 \neq null \wedge owns(z_1, z_2.lock)) \rightarrow (z_1.started \wedge z_2 = z_1))
\end{aligned}
$$

$$
\begin{aligned}
&I_{Monitor} \stackrel{def}{=}\\
&\quad (\forall(e \in wait \cup notified).e = (creator, 1)) \wedge\\
&\quad (x = 0 \rightarrow (lock = (null, 0) \wedge length(wait) = 0 \wedge length(notified) = 0 \wedge \neg started)) \wedge\\
&\quad (x = 1 \rightarrow (lock = (creator, 1) \wedge length(wait) = 0 \wedge length(notified) = 0 \wedge \neg started)) \wedge\\
&\quad ((x = 2 \vee x = 7) \rightarrow\\
&\qquad (lock = (creator, 1) \wedge length(wait) = 0 \wedge length(notified) = 0 \wedge started)) \wedge\\
&\quad (x = 3 \rightarrow (lock = (null, 0) \wedge length(wait) = 1 \wedge length(notified) = 0 \wedge started)) \wedge\\
&\quad (x = 4 \rightarrow (lock = (this, 1) \wedge ((length(wait) = 1 \wedge length(notified) = 0) \vee\\
&\qquad\qquad (length(wait) = 0 \wedge length(notified) = 1)) \wedge started)) \wedge\\
&\quad (x = 5 \rightarrow (lock = (this, 1) \wedge length(wait) = 0 \wedge length(notified) = 1 \wedge started)) \wedge\\
&\quad (x = 6 \rightarrow (lock = (null, 0) \wedge length(wait) = 0 \wedge length(notified) = 1 \wedge started)) \wedge\\
&\quad (x = 8 \rightarrow (lock = (null, 0) \wedge length(wait) = 0 \wedge length(notified) = 0 \wedge started))
\end{aligned}
$$

```
class Main{
    ⟨ Int x; ⟩
    ⟨ Monitor created; ⟩

    nsync Void wait(){ {false}^?call {false} return_getlock {false}^!ret }

    nsync Void run(){
        Monitor obj;
        {x = 0 ∧ thread = this ∧ conf = 0 ∧ started}
        obj := new^Monitor; {thread = this ∧ conf = 0}^new ⟨created, x := obj, 1⟩^new
```

$\{x = 1 \wedge \mathit{thread} = \mathit{this} \wedge \mathit{conf} = 0 \wedge \mathit{started} \wedge \mathit{created} = \mathit{obj} \wedge \mathit{obj} \neq \mathit{null}\}$
`obj.m1()`
$\{x = 1 \wedge \mathit{thread} = \mathit{this} \wedge \mathit{conf} = 0 \wedge \mathit{created} = \mathit{obj}\}^{!call}$
$\langle x := (\textsf{if } \mathit{obj} = \mathit{this} \textsf{ then } x \textsf{ else } 2 \textsf{ fi})\rangle^{!call}$
$\{x = 2 \wedge \mathit{thread} = \mathit{this} \wedge \mathit{conf} = 0 \wedge \mathit{created} = \mathit{obj}\}^{wait}$
$\{x = 2 \wedge \mathit{thread} = \mathit{this} \wedge \mathit{conf} = 0 \wedge \mathit{created} = \mathit{obj}\}^{?ret}$
$\langle x := (\textsf{if } \mathit{obj} = \mathit{this} \textsf{ then } x \textsf{ else } 3 \textsf{ fi})\rangle^{?ret}$
$\{x = 3 \wedge \mathit{thread} = \mathit{this} \wedge \mathit{conf} = 0 \wedge \mathit{created} = \mathit{obj}\}$
`}`
`}`

`class Monitor{`
$\langle$ `Main creator;` $\rangle$
$\langle$ `Int x;` $\rangle$

`nsync Void wait(){`
$\{x = 2 \wedge \mathit{thread} = \mathit{creator}\}^{?call}$ $\langle x := 3\rangle^{?call}$
$\{3 \leq x \wedge x \leq 6 \wedge \mathit{thread} = \mathit{creator}\}$
$\texttt{return}_{getlock}$
$\{x = 6 \wedge \mathit{thread} = \mathit{creator}\}^{!ret}$ $\langle x := 7\rangle^{!ret}$
`}`

`nsync Void notify(){`
$\{x = 4 \wedge \mathit{thread} = \mathit{this} \wedge \mathit{length}(\mathit{wait}) = 1\}$
$\langle\rangle$
$\{x = 4 \wedge \mathit{thread} = \mathit{this} \wedge \mathit{length}(\mathit{wait}) = 0\}$
`return`
$\{x = 4 \wedge \mathit{thread} = \mathit{this} \wedge \mathit{length}(\mathit{wait}) = 0\}^{!ret}$ $\langle x := 5\rangle^{!ret}$
`}`

`nsync Void notifyAll(){`
$\{\mathit{false}\}$ $\langle\rangle$ $\{\mathit{false}\}$
`}`

`sync Void m1(){`
$\{x = 0\}^{?call}$ $\langle \mathit{creator} := \mathit{thread}; x := 1\rangle^{?call}$
$\{x = 1 \wedge \mathit{thread} = \mathit{creator} \wedge \mathit{conf} = 0\}$
`start();`
$\{x = 2 \wedge \mathit{thread} = \mathit{creator}\}$
`wait();`
$\{x = 7 \wedge \mathit{thread} = \mathit{creator}\}$
`return`
$\{x = 7 \wedge \mathit{thread} = \mathit{creator}\}^{!ret}$ $\langle x := 8\rangle^{!ret}$
`}`

`nsync Void run(){`
$\{x = 1 \wedge \mathit{thread} = \mathit{this} \wedge \textsf{caller} = (\mathit{this}, 0, \mathit{creator})\}^{?call}$ $\langle x := 2\rangle^{?call}$
$\{(x = 2 \vee x = 3) \wedge \mathit{thread} = \mathit{this} \wedge \mathit{started}\}$
`m2()`
$\{(x = 6 \vee x = 7 \vee x = 8) \wedge \mathit{thread} = \mathit{this}\}$
`}`

`sync Void m2(){`
$\{x = 3 \wedge \mathit{thread} = \mathit{this}\}^{?call}$ $\langle x := 4\rangle^{?call}$
$\{x = 4 \wedge \mathit{thread} = \mathit{this} \wedge \mathit{length}(\mathit{wait}) = 1 \wedge \mathit{started}\}$
`notify();`
$\{x = 5 \wedge \mathit{thread} = \mathit{this}\}$
`return`
$\{x = 5 \wedge \mathit{thread} = \mathit{this}\}^{!ret}$ $\langle x := 6\rangle^{!ret}$
`}`
`}`

B.3 A producer-consumer example

$GI \stackrel{def}{=}$
$(\forall(p : Producer).(p \neq null \wedge \neg p.outside \wedge p.consumer \neq null) \rightarrow$
$(p.consumer.lock = (null, 0) \wedge length(p.consumer.wait) = 0 \wedge$
$p.consumer.producer = null \wedge \neg p.consumer.started \wedge p.consumer.counter = 0)) \wedge$
$(\forall(p : Producer).(p \neq null \wedge p.consumer \neq null \wedge p.consumer.producer \neq null) \rightarrow$
$p.outside) \wedge$
$(\forall(c : Consumer).(c \neq null \wedge c.started) \rightarrow (c.producer \neq null \wedge c.producer.started)) \wedge$
$(\forall(c1, c2 : Consumer).(c1 \neq null \wedge c2 \neq null) \rightarrow c1 = c2)) \wedge$
$(\exists(p : Producer).p \neq null \wedge (\forall(p2 : Producer).p2 \neq null \rightarrow p2 = p) \wedge$
$(p.consumer = null \rightarrow (\forall(c : Consumer).c = null))) \wedge$
$(\forall(c : Consumer).(c \neq null \wedge c.producer \neq null) \rightarrow c.producer.started)$

$I_{\mathsf{Consumer}} \stackrel{def}{=}$
$(lock = (null, 0) \vee (owns(this, lock) \wedge started) \vee owns(producer, lock)) \wedge length(wait) \leq 1$

```
class Producer{
    ⟨ Consumer consumer; ⟩
    ⟨ Bool outside; ⟩

    nsync Void wait(){ {false}^?call {false} return_getlock {false}^!ret }

    nsync Void run(){
        Consumer c;
        {¬outside ∧ thread = this ∧ consumer = null ∧ started}
        c := new^Consumer; {thread = this}^new ⟨consumer := c⟩^new
        {c = consumer ∧ ¬outside ∧ consumer ≠ null ∧ consumer ≠ this∧
          thread = this ∧ started}
        c.produce() {thread = this}^!call
            ⟨outside := (if c = this then outside else true)⟩^!call
        {false}
    }
}

class Consumer{
    Int buffer;
    ⟨ Producer producer; ⟩

    nsync Void wait(){
        {owns(thread, lock) ∧ started ∧ length(wait) = 0}^?call
        {started ∧ not_owns(thread, lock) ∧ (thread = this ∨ thread = producer)∧
          (thread ∈ wait ∨ thread ∈ notified)}
        return_getlock
        {started ∧ lock = (null, 0) ∧ thread ≠ null ∧ (thread = this ∨ thread = producer)∧
          thread ∈ notified}^!ret
    }

    nsync Void notify(){
        {owns(thread, lock) ∧ started}
        ⟨⟩
        {owns(thread, lock) ∧ length(wait) = 0}
    }

    nsync Void notifyAll(){ {false} ⟨⟩ }

    sync Void produce(){
        Int i;
        {thread ≠ null ∧ producer = null ∧ thread = proj(caller, 1)∧
          length(wait) = 0 ∧ ¬started}^?call
        ⟨producer := proj(caller, 1)⟩^?call
```

```
        {owns(thread, lock) ∧ thread = producer ∧ ¬started ∧ conf = 0 ∧ producer ≠ this}
        i:=0;
        {owns(thread, lock) ∧ thread = producer ∧ ¬started ∧ conf = 0 ∧ producer ≠ this}
        start();
        {owns(thread, lock) ∧ started ∧ thread = producer}
        while (true) do
            {owns(thread, lock) ∧ started ∧ thread = producer}
            //produce i here
            buffer := i;
            {owns(thread, lock) ∧ started ∧ thread = producer}
            notify();
            {started ∧ thread = producer}^wait
            {owns(thread, lock) ∧ started ∧ thread = producer ∧ length(wait) = 0}
            wait()
            {started ∧ thread = producer}^wait
            {owns(thread, lock) ∧ started ∧ thread = producer}
        od;
        {false}
        return
        {false}^!ret
    }

    nsync Void run(){
        {¬started ∧ caller = (this, 0, producer)}^?call
        {not_owns(thread, lock) ∧ thread = this ∧ thread ≠ null ∧ started}
        consume()
        {false}
    }

    sync Void consume(){
        Int i;

        {thread = this ∧ free_for(thread, lock) ∧ started}^?call
        {owns(thread, lock) ∧ started ∧ thread = this}
        while (true) do
            {owns(thread, lock) ∧ started ∧ thread = this}
            i := buffer;
            //consume i here
            {owns(thread, lock) ∧ started ∧ thread = this}
            notify();
            {started ∧ thread = this}^wait
            {owns(thread, lock) ∧ started ∧ thread = this ∧ length(wait) = 0}
            wait()
            {thread = this}^wait
            {owns(thread, lock) ∧ started ∧ thread = this}
        od;
        {false}
        return
        {false}^!ret
    }
}
```

Summary

The aim of program verification is to prove that a program running on a computer does exactly what one expects. In this thesis we focus on programs written in (a subset of) the programming language *Java*, but the results can be adapted also to other languages with similar features.

The development of a verification technique goes through several stages: First, for a programming language with a given syntax we have to formalize its semantics, i.e., its meaning. That means, we give a *precise meaning* to the *Java* programs considered in this dissertation, without allowing ambiguities.

Next we have to define a logic which allows to formalize *properties* of programs written in that language. That is, we introduce another (formalized) language, with an equally precise semantics, in which we express properties which should be satisfied by the *Java* programs. In our case, the underlying logic is a superset of first-order predicate logic.

Then we define a proof system, which describes general conditions which assure that some given properties are satisfied by a given program. In this thesis we restrict these properties to *invariants*, i.e., properties which should hold during the entire execution of a program. So, we do not focus on properties which, e.g., express that a computation reaches certain locations (repeatedly), the so-called lifeness properties.

To prove that a program property is an invariant, first we have to specify the required property in the logic. Then we have to apply the proof system to the program, which results in a set of verification conditions, i.e., logical implications which should hold for that property to be an invariant of the program considered. The characterizing feature of those properties is that they should hold in the underlying logic. More precisely, although these verification conditions are formulated depending on the particular program considered and the particular specification of which one wants to prove that the program satisfies it, they should hold in the underlying predicate logic, only. Thus, program verification is reduced to proving a finite number of properties in the underlying logic.

We have implemented the tool *Verger* which automatically generates the verification conditions for an input program with its specification. Validity of these conditions assures that the program property is invariant. We use the theorem prover *PVS* to prove these verification conditions.

The *Java* language is a very large programming language, i.e., it has a lot of different programming constructs and interesting features. In this thesis we consider a small subset of *Java* only, focusing on its concurrency features. And even for that subset, it is hard to exactly describe the behavior of the corresponding *Java* programs, because there are so many semantic trouble spots left in *Java* that their full semantics would amount only to a precise description of the meaning of programs using a particular compiler in a particular context and the like. Clearly, that is undoable for a language as large as *Java*, and also undesirable. Our intention is to focus on those aspects of *Java* concurrency which are deemed to be generally understood, that is, depending on the source code of concurrent *Java* programs, only, and not on any implementation feature or environment property. To do so, we give an abstract semantics for the *Java* programs considered, although, as remarked above, this does not necessarily imply a full correspondence with their implementation behavior.

To transparently describe the proof system, we present it incrementally in three stages, starting with a minimal language and later adding new language features. We start in Chapter 2 with a proof method for a *sequential* sublanguage of *Java*, where each program is executed by a single process, a so-called *thread*. In the second stage in Chapter 3 we additionally allow dynamic thread creation, leading to *multithreaded* execution. Finally, we integrate *Java*'s *monitor synchronization* mechanism in Chapter 4. Monitor synchronization allows special coordination between threads. This construct is usually used to assure mutual exclusion, i.e., to exclude the possibility that different threads have simultaneous access to some resource like, for example, shared memory.

This incremental development shows how the proof system can be extended stepwise to deal with additional features of the programming language. Further extensions by, for example, the concepts of inheritance and subtyping are topics for future work (see Section 8).

This dissertation offers *soundness* and *(semantic) completeness* proofs for our proof system. Soundness of a proof system means, that if a program with its specification satisfies the requirements imposed by the proof system, i.e., the verification conditions generated, then the specification is, indeed, always an invariant property of the program, i.e., it holds during program execution. In practice this means that using our proof system we can only derive properties which, indeed, are true during the execution of the program considered. In short: we cannot prove nonsense.

Completeness on the other hand means, that if a program satisfies an invariant property, then this fact is always provable with the help of the proof system. Soundness and completeness of the proof method for the third language is discussed in Chapter 6; the proofs can be found in the appendix. Further possible extensions of the proof system to cover additional programming language features are discussed in Chapter 8.

Finally, as mentioned earlier, to prove correctness of program properties one has to apply the proof system to the given *Java* program together with its specification. This process results in a set of verification conditions, which must be proven. We have developed the *Verger* tool as computer support for this task. The tool takes a program with its specification as input and generates the verification conditions, which assure invariance of the specification, in the syntax of the theorem prover *PVS*. This theorem prover is finally used to verify the conditions. Computer support is described and illustrated by some examples in Chapter 9.

Samenvatting

In dit proefschrift wordt voor de eerste keer beschreven hoe men eigenschappen van parallelle *Java* programma's met wiskundige precisie kan afleiden. In feite is het verbazingwekkend dat dit niet veel eerder is gebeurd. Want *Java* is sinds 1996 in zwang als populaire programmeertaal, en *Java* is bij uitstek de taal waarin heden ten dage betrouwbare 'server farms' geprogrammeerd worden. D.w.z., parallelle *Java* programma's zijn gemeengoed in onze programmeercultuur en worden gebruikt om systemen te programmeren. De naïeve leek denkt wellicht dat het 'vanzelfsprekend' is dat zulke programma's foutloos functioneren. Wel, dat is niet het geval. Alleen werkelijk topklasse *Java* programmeurs foutloos te programmeren.

Het onderliggende probleem is dat *Java* zeer complex en zeker geen betrouwbare programmeertaal is. Er is veel deskundigheid voor nodig om in die deelverzameling van *Java* te programmeren welke tot enigermate voorspelbaar gedrag van de desbetreffende programma's leidt.

Aangezien het moeilijk is onvoorspelbaar gedrag wiskundig vast te leggen zonder 'het kind met het badwater weg te gooien', leidt deze problematiek tot de eerste opgave die in dit proefschrift opgelost wordt: hoe een zinvolle deelverzameling van *Java* programma's te definiëren, waarvan het gedrag zowel voorspelbaar is als algemeen geaccepteerd wordt. Dit leidt tot de definitie van $Java_{synch}$.

Onze tweede opgave is de betekenis van die deelverzameling, d.w.z., van $Java_{synch}$, wiskundig vast te leggen. Want alleen met behulp van wiskunde zijn onomstootbare uitspraken af te leiden.

De derde opgave is het vastleggen van de taal waarin wij eigenschappen van $Java_{synch}$ programma's kunnen formuleren, en waarmee wij deze kunnen afleiden. D.w.z., het handelt zich hier niet alleen om een passende formele taal maar ook om de bijbehorende logica.

Hieruit volgt dan meteen onze vierde opgave: hoe leiden we af dat $Java_{synch}$ programma's aan in die (specificatie-)taal geformuleerde eigenschappen voldoen?

De vijfde opgave heeft een meer academisch karakter: enerzijds te bewijzen dat het door ons geformuleerde afleidingssysteem alleen zodanige eigenschappen laat afleiden die ook inderdaad gelden voor de desbetreffende $Java_{synch}$ programma's en, anderzijds, dat alle in onze specificatietaal te formuleren geldige eigenschappen van $Java_{synch}$ programma's in ons systeem afleidbaar zijn.

Deze vijf opgaven worden in dit proefschrift opgelost, en met behulp van uitgewerkte voorbeelden geïllustreerd. Omdat aan dit proces zeer vele details kleven, ligt het voor de hand hier de rekenmachine zelf bij in te schakelen. In ons geval heeft dit geleid tot het programmeren van een softwarepakket dat ons bij het afleiden van eigenschappen van $Java_{synch}$ programma's ten dienste staat, het zogenaamde *Verger* pakket.

Curriculum Vitae

Contact Information

Name	Erika Ábrahám
Address	Albert-Ludwigs-Universität Freiburg Institut für Informatik Georges-Köhler-Allee 52 D-79110 Freiburg i.Br. Germany
Phone	+49-761-203 8251
Fax	+49-761-203 8242
Email	eab@informatik.uni-freiburg.de
Home page	http://www.informatik.uni-kiel.de/∼ eab/

Personal Details

Date of Birth	11/09/1970
Nationality	Hungarian
Sex	female
Marital status	divorced
Children	Judith (7) and András (5)

Education

1984-1989	Radnóti Miklós Gimnázium, Szeged, Hungary
1990	Different German language courses
1991	Acquisition of the German matriculation standard
1992-1999	Christian-Albrechts-University Kiel, Germany
Degree	Master of Computer Science
Major	Computer Science
Minor	Theoretical Physics
Average mark	1.0 (award)
Master's thesis	Head-pose estimation from facial images with Subspace Neural Networks

Estimating the pose of human heads from camera images is an important task in, e.g., driver surveillance and the design of advanced human-machine interfaces. We used subspace neural networks to solve this task [22, 21, 20].
Generally speaking, neural networks are systems which are able to "learn" to react to certain inputs with certain outputs, i.e., to learn a function mapping values from an input space to values from an output space. Learning is based on a learning algorithm and a training set consisting of input-output value pairs. The neural network gets trained to learn a function approximating the mapping of the input values of the training set to the corresponding output values. After training, the neural network accepts inputs from the whole input space and computes its output using the learned approximation.
To train neural networks to estimate head-poses from camera images, a roboter arm has turned a doll in different poses, and we recorded camera images from each pose. We extracted characteristic information from the images; the extracted information, paired with the vectors specifying the corresponding pose, served as the training set for neural networks. Most of the networks could compute the pose of the doll face with an average error under 1° per dimension.

Since 1999	Research assistant and Ph.D. student at the Christian-Albrechts-University Kiel, Germany
Since 2002	Guest research fellow at the Albert-Ludwigs-University Freiburg, Germany

My first research topic during my Ph.D. period was the formalization and implementation of an assertional proof method for hybrid systems [16, 17, 18]. Hybrid systems are a mathematical model to describe discrete systems acting in a continuous environment. Since such systems are increasingly used in safety-critical applications, the development of verification techniques is crucial.
While most work in this area is done in the field of model checking, less attention has been paid to deductive techniques. We have developed a deductive assertional proof method for the analysis of hybrid systems and their parallel composition.

The syntax and the semantics of hybrid systems and their parallel composition, as well as the proof system and a number of examples, are implemented using the theorem prover *PVS*. Soundness of the proof system has been proven using the theorem prover.

Later, I started to deal with *Java* and its proof theory, which constitutes the topic of this thesis.

Experience

1989-1990	Employed in Hungary (Computer center of PICK).
1993-1996	Scientific collaborator at the University Library of Kiel: Development and maintenance of a CD-ROM database and a campus network for scientific literature investigation.
1993-1994	Scientific collaborator in the project “Computer at the school” (University Kiel, Institute for Science Education).
1995-1996	Scientific collaborator in a project dealing with the development of a modern teaching technique for natural sciences in the school (University Kiel, Institute for Science Education).
1997-1999	Tutor at the Department of Computer Science and Applied Mathematics, University of Kiel, Germany: Supervision of students, presentation of tutorials.
Since 1999	Research in the context of national and European projects; teaching. Focus on the deductive computer-supported verification of discrete and continuous systems (object-oriented concurrent languages, hybrid systems).

Further Achievements

- University Kiel: Award for the achievements in master thesis and examination.
- Best presentation award at ICECCS'01.
- Invited talk at FMCO'02.

Skills

Languages	Hungarian (native), German (fluent), English (fluent).
Other skills	Playing the piano.

Publications

[1] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. An assertion-based proof system for multithreaded Java. *Theoretical Computer Science*, 2004. to appear.

[2] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof outlines for exceptions in multithreaded Java. 2004. Submitted for publication, June 2004.

[3] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof outlines for multithreaded Java with exceptions. Technical Report 0313, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, December 2003. Available at `http://www.informatik.uni-kiel.de/reports/2003/0313.html`.

[4] Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, and Martin Steffen. Object connectivity for a concurrent class calculus (extended abstract). 2004. Submitted for publication. A preliminary and longer version appeared under the title "A Structural Operational Semantics for a Concurrent Class Calculus" as Technical Report 0307, CAU, Institute of Computer Science August 2003.

[5] Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, and Martin Steffen. A structural operational semantics for a concurrent class calculus. Technical Report 0307, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, August 2003.

[6] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof-outlines for monitors in Java. In Najm et al. [23], pages 155–169. A longer version appeared as technical report TR-ST-03-1, April 2003 (`http://www.informatik.uni-kiel.de/inf/deRoever/techreports/03/tr-st-03%-1.pdf`).

[7] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A Hoare logic for monitors in Java. Techical report TR-ST-03-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, April 2003. Available

at `http://www.informatik.uni-kiel.de/inf/deRoever/techreports/03/tr-st-03-%1.pdf`.

[8] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A tool-supported assertional proof system for multithreaded Java. In Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, and Erik Poll, editors, *Proc. of the Workshop on Formal Techniques for Java-like Programs - FTfJP'2003*, 2003.

[9] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A tool-supported proof system for monitors in Java. In Bonsangue et al. [24], pages 1–32.

[10] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A compositional operational semantics for Java_{MT}. In Derschowitz [25], pages 290–303. A preliminary version appeared as Technical Report TR-ST-02-2, May 2002.

[11] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A compositional operational semantics for Java_{MT}. Technical Report TR-ST-02-2, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, May 2002.

[12] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In Nielsen and Engberg [26], pages 4–20. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.

[13] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept: Soundness and completeness. Technical Report TR-ST-02-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 2002.

[14] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Deductive verification for multithreaded Java (extended abstract). In *Proceedings of the "11. Kolloquium Programmiersprachen und Grundlagen der Programmierung", 2001, Rurberg*, pages 121–126, 2001.

[15] Erika Ábrahám-Mumm and Frank S. de Boer. Proof-outlines for threads in Java. In Palamidessi [27], pages 229–242.

[16] Erika Ábrahám-Mumm, Ulrich Hannemann, and Martin Steffen. Assertion-based analysis of hybrid systems with PVS. In Moreno-Díaz and Buchberger [28].

[17] Erika Ábrahám-Mumm, Ulrich Hannemann, and Martin Steffen. Verification of hybrid systems: Formalization and proof rules in PVS. In *Proceedings of the Seventh IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2001)*, 2001. A preliminary and longer version appeared as technical report TR-ST-01-1.

[18] Erika Ábrahám-Mumm, Ulrich Hannemann, and Martin Steffen. Verification of hybrid systems: Formalization and proof rules in PVS. Technical Report TR-ST-01-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, January 2001.

[19] Jan B. de Meer and Erika Ábrahám-Mumm. Formal methods for reflective system specification. In Grabowski and Heymer [29], pages 51–57.

[20] Erika Ábrahám-Mumm. Bestimmung der Gesichtspose mit künstlichen neuronalen Netzen. Master's thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, April 1998.

[21] Jörg Bruske, Erika Ábrahám-Mumm, Joseph Pauli, and Gerald Sommer. Head-pose estimation from facial images with subspace neural networks. In *1998 Int. Conf. on Neural Network and Brain Proc. (ICNN&B'98)*, pages 528–530. Publishing House of Electronics Industry, 1998.

[22] Jörg Bruske, Erika Ábrahám-Mumm, and Gerald Sommer. Visuomotorische Koordination eines Roboterarmes mit Kohonen-Karten, Neuronalem Gas und Dynamischen Zellstrukturen - Ein Vergleich. In *Proc. Selbstorganisation von Adaptivem Verhalten 1997 (SOAVE'97)*, Vortschrittsberichte VDI, Reihe 8, Nr. 663, pages 203–211. VDI Verlag, 1997.

[23] Elie Najm, Uwe Nestmann, and Perdita Stevens, editors. *Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '03), Paris*, volume 2884 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2003.

[24] Marcello M. Bonsangue, Frank S. de Boer, Willem-Paul de Roever, and Susanne Graf, editors. *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002), Leiden*, volume 2852 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[25] Nachum Derschowitz, editor. *Proceedings of the International Symposium on Verification (Theory and Practice), Celebrating Zohar Manna's 64th Birthday, Taormina, Sicily, June 29–July 4, 2003*, volume 2772 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[26] Mogens Nielsen and Uffe H. Engberg, editors. *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2002), Held as Part of the Joint European*

Conferences on Theory and Practice of Software (ETAPS 2002), (Grenoble, France, April 8-12, 2002), volume 2303 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2002.

[27] Catuscia Palamidessi, editor. *CONCUR 2000: Concurrency Theory (11th International Conference, University Park, PA, USA)*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2000.

[28] Roberto Moreno-Díaz and Bruno Buchberger, editors. *Computer Aided Systems Theory (EUROCAST 2001), A Selection of Papers from the 8th International Workshop on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, February 19–23, 2001.*, volume 2178 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[29] Jens Grabowski and Stefan Heymer, editors. *Formale Beschreibungstechniken für verteilte Systeme*. Universität Lübeck/Shaker Verlag, Aachen, Juni 2000.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms.* Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation.* Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search.* Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory.* Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems.* Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance.* Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual.* Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits.* Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism.* Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference.* Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics.* Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth.* Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context.* Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types.* Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics.* Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution.* Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing.* Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing.* Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering.* Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation.* Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller.* Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model.*

Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences.* Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes.* Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes.* Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking.* Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols.* Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents.* Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ.* Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking.* Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL.* Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01