A Deductive Proof System for Multithreaded Java with Exceptions

Erika Ábrahám
Albert-Ludwigs-University Freiburg, Germany

Frank S. de Boer
CWI, Amsterdam, The Netherlands

Willem-Paul de Roever
Christian-Albrechts-University, Kiel, Germany

Martin Steffen
Christian-Albrechts-University, Kiel, Germany

Abstract. Besides the features of a class-based object-oriented language, Java integrates concurrency via its thread-classes, allowing for a multithreaded flow of control. Besides that, the language offers a flexible exception mechanism for handling errors or exceptional program conditions.

To reason about safety-properties of Java-programs and extending previous work on the proof theory for monitor synchronization, we introduce in this paper an assertional proof method for Java_{MT} ("Multi-Threaded Java"), a small concurrent sublanguage of Java, covering concurrency and especially exception handling. We show soundness and relative completeness of the proof method.

1. Introduction

Since the Java language is increasingly used also in safety-critical applications, the development of verification techniques for Java programs becomes more and more 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 work introduces a tool-supported assertional proof method for a concurrent sublanguage of Java. The language includes dynamic object creation, method invocation, object references with aliasing, concurrency, Java’s monitor discipline, and exception handling, but excludes inheritance and subtyping. The concurrency model includes shared-variable concurrency via instance variables, coordination via reentrant synchronization monitors, synchronous message passing, and dynamic thread creation.

To support a clean interface between internal and external object behavior, we exclude qualified references to instance variables. I.e., the values of instance variables of an object can be accessed and

*Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-{1,2,4}).
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.

In order to capture program behavior in a modular way, 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. The global behavior, including the communication topology of objects, is expressed in the global language. As in the Object Constraint Language (OCL) [57], properties of object-structures are described in terms of a navigation or dereferencing operator.

The assertional proof system is formulated in terms of proof outlines [44], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [24, 25]. 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 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 [44, 33], formulated also in the local language. It has especially to accommodate for reentrant code and the specific synchronization mechanism. Possibly affecting more than one instance, communication and object creation is treated in the cooperation test, using the global language. The communication 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 [14] and in [33] for CSP.

Our proof method is modular in the sense that it allows for separate interference freedom and cooperation tests. 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 released, without loosing the mentioned modularity, but it would increase the complexity of the proof system. 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 [45] to verify the conditions, for which we only need to encode the semantics of the assertion language.

To transparently describe the proof system, we present it incrementally in four 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 thread of execution. In the second stage 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. We integrate Java’s monitor synchronization mechanism in the third stage. Finally, we include Java’s exception handling in the last stage. The proof system is sound and complete.

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.

1.1. Related work

This work extends earlier results. In [7] we develop a proof system for a concurrent sublanguage of Java, but without reentrant monitors. Reentrant synchronization was incorporated in [10]; the work [2]
1.1 Related work

integrates also Java’s monitor methods wait, notify, and notifyAll. An incremental description of the proof system, starting with a sequential language and stepwise adding additional language features, but excluding exception handling, is given in [9]. In [9] we also introduce proof conditions for deadlock freedom. The work is summarized in Ábrahám’s PhD thesis [1] and the theoretical aspects in [6]. We discuss the proof system also in [8] and in [4]. We formalize the semantics of our programming language in a compositional manner in [5]. This work extends the above ones by including exception handling.

The semantical foundations of Java have been thoroughly studied ever since the language gained widespread popularity (see e.g. [11, 55, 23]). The research concerning Java’s proof theory mainly concentrated on various aspects of sequential sub-languages. To the best of our knowledge, our work defines the first sound and complete assertional proof method for a multithreaded sublanguage of Java including its monitor discipline and exception handling.

De Boer [19] presents a sound and complete proof system in a weakest precondition formulation for a parallel object-based language, i.e., without inheritance and subtyping, and also without reentrant method calls. Later work [48, 21, 20] and especially the PhD thesis of Pierik [47] includes more features, especially catering for a Hoare logic for inheritance and subtyping.

The aim of the work in the LOOP project (Logic of Object-Oriented Programming) [34] is to specify and verify properties of classes in object-oriented languages. The project research concentrates on a sequential subpart of Java; the main focus of application is JavaCard. A compiler [17] translates programs and their specifications into PVS [30] and Isabelle/HOL [16]. The translation is based on the embedding of a denotational semantics of the sequential Java subset into Higher Order Logic (HOL). Soundness of the representation is shown in [26]. LOOP specifications formalized in JML are represented in HOL by a set of proof rules [32]. Jacobs presents also a coalgebraic view of exceptions in [29]. Modeling inheritance in higher order logic is the topic of [27]. The LOOP tool and methodology has been applied to several case studies; see e.g. [54, 53, 18, 28, 31].

Instead of the denotational semantics, our work is based on an operational semantics. Though research within the LOOP project deals with many of the complexities of Java, they don’t handle concurrency, and don’t investigate completeness.

The project Bali [15] is concerned with the formalization of various aspects of Java in the theorem prover Isabelle/HOL [46]. Nipkow and von Oheimb [37, 42] prove type soundness of their Java\textsubscript{light} subset, a large sequential sublanguage of Java. They formalize its abstract syntax, type system, and well-formedness conditions. Instead of the denotational semantics in works of the LOOP project, they 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\textsubscript{light}, von Oheimb presents an axiomatic semantics [39, 40], and proves soundness and completeness of the latter with respect to the operational semantics. With µJava, Nipkow et al. [38] 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 [38], von Oheimb [41] presents a Hoare-style calculus for a JavaCard subset and proves soundness and completeness in Isabelle/HOL. Nipkow [36] selects some of the technically difficult language features and deals with their Hoare logic in isolation. The combination of [41] and [36] in one language (NanoJava) is formulated in [43].

In contrast to our approach, the Bali project aims to cover only sequential subsets of Java. Furthermore, they use a semantic representation of assertions; program execution is specified by state transformations. Our proof system uses a syntactic representation, and substitution operators instead of state
transformations.

Similarly to our proof system, also Poetzsch-Heffter and Müller use a syntactical representation of assertions [49, 50, 51, 52]. They 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.

1.2. Overview

The work is organized as follows: Section 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. Section 3 extends the results to a concurrent sublanguage. The language introduced in Section 4 includes Java’s monitor synchronization mechanism. Section 5 covers also exception handling. Soundness and completeness are discussed in Section 6. Section 7 contains some concluding remarks.

2. The sequential language

In this section we introduce a sequential sublanguage $\text{Java}_{seq}$ of Java. We define its syntax in Section 2.1, and its semantics in Section 2.2. After defining the assertional language in Section 2.3, we introduce a proof system for verifying safety properties of the language in Section 2.4.

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 $\text{Java}_{seq}$ 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 is 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 $\text{Java}_{seq}$.

2.1. Syntax

$\text{Java}_{seq}$ is a strongly typed language; besides class types $c$, it supports booleans $\text{Bool}$ and integers $\text{Int}$ as primitive types, and pairs $t \times t$ and lists $\text{list} t$ as composite types. The type of methods without return value is $\text{Void}$. Since $\text{Java}_{seq}$ 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 $t_1 \times \cdots \times t_n \rightarrow t$ 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.

We notationally distinguish between instance variables $x \in \text{IVar}$ and local (temporary) variables $u \in \text{TVar}$. 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 use \( \mathit{Var} = \mathit{IVar} \cup \mathit{TVar} \) for the set of program variables with typical element \( y \), where \( \cup \) is the disjoint union operator.

The abstract syntax is summarized in Table 1. It slightly differs from \textit{Java} syntax. Though we use the abstract syntax for the theoretical part of this work, our tool supports \textit{Java} syntax.

Besides using instance and local variables, expressions \( \mathit{exp} \in \mathit{Exp} \) are built from the self-reference \textit{this}, the empty reference \textit{null}, and from subexpressions using the given operators. We use \( e \) as typical element for expressions. To support a clean interface between internal and external object behavior, \textit{Java}_{seq} does not allow qualified references to instance variables. 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 \( \mathit{stm} \in \mathit{Stm} \), we allow assignments, object creation, method invocation, and standard control constructs like sequential composition, conditional statements, and iteration. We write \( \epsilon \) for the empty statement.

A method definition \( m(u_1, \ldots, u_n)\{ \mathit{stm}; \mathit{return} \ \mathit{e_{ret}} \} \) specifies the method’s name \( m \), a list of formal parameters \( u_1, \ldots, u_n \), and a method body of the form \( \mathit{stm}; \mathit{return} \ \mathit{e_{ret}} \), i.e., we require that method bodies are terminated by a single return statement, giving back the control and possibly a return value. The set \( \mathit{Meth}_c \) contains the methods of class \( c \). We denote the body of method \( m \) of class \( c \) by \( \text{body}_{m,c} \).

Sometimes we explicitly mention the types of formal parameters and of the return value in \textit{Java}-style \( t \ m(t_1 u_1, \ldots, t_n u_n)\{ \text{body}_{m,c} \} \).

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 \( \mathit{class}_{\mathit{main}} \) 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.\footnote{In \textit{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 \textit{Java}, we assume that the main class is a \texttt{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 \textit{Java}’s main-method is static, but our proof system does not support static methods and variables.} 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 explicitly define variables in Java-style.

Besides the mentioned simplifications on 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 stored in instance variables. This restriction allows for a proof system with separated verification conditions for interference freedom and cooperation. It should be clear that it is possible to transform a program to adhere to this restrictions at the expense of additional local variables and thus new interleaving points. The above restrictions could be released, without loosing the mentioned modularity, but it would increase the complexity of the proof system.

2.2. Semantics

In this section, we define the operational semantics of $Java_{seq}$. After introducing the semantic domains, we describe states and configurations. The operational semantics is presented by transitions between program configurations.

2.2.1. States and configurations

Let $Val^t$ be the disjoint domains of the various types $t$. For class names $c$, the disjunct sets $Val^c$ with typical elements $\alpha, \beta, \ldots$ denote infinite sets of object identifiers. The value of null of type $c$ is $null^c \notin Val^c$. In general we will just write null, when $c$ is clear from the context. We define $Val^c_{null}$ as $Val^c \cup \{null^c\}$, and correspondingly for compound 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 compound types, where sequences are initially empty. We define 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 and 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 $TVar \rightarrow Val_{null}$. We refer to local states of method $m$ of class $c$ by $\tau^{m,c}$. The initial local state $\tau^{init}_{m,c}$ assigns to each local variable $u$ from its domain the value $Init(u)$. A local configuration $(\alpha, \tau, stm)$ of a method of an object $\alpha \neq null$ specifies, in addition to its local state $\tau$, its point of execution represented by the statement $stm$. A thread configuration $\xi = (\alpha_0, \tau_0, stm_0)(\alpha_1, \tau_1, stm_1) \ldots (\alpha_n, \tau_n, stm_n)$ is a stack of local configurations, representing the

\footnote{In Java, this is a “final” instance variable, which for instance implies, it cannot be assigned to.}
chain of method invocations of the given thread. We write \( \xi \circ (\alpha, \tau, stm) \) for pushing a new local configuration onto the stack.

Objects are characterized by their instance states \( \sigma_{\text{inst}} \in \Sigma_{\text{inst}} \) of type \( IVar \cup \{\text{this}\} \rightarrow Val_{\text{null}} \); we require that this is in the domain \( \text{dom}(\sigma_{\text{inst}}) \) of \( \sigma_{\text{inst}} \). We write \( \sigma_{\text{inst}}^{\text{c}} \) to denote states of instances of class \( c \). The semantics will maintain \( \sigma_{\text{inst}}^{\text{c}}(\text{this}) \in Val^{c} \) as invariant. The initial instance state \( \sigma_{\text{inst}}^{\text{c}} \) assigns a value from \( Val^{c} \) to this, and to each of its remaining instance variables \( x \) the value \( \text{Init}(x) \).

A global state \( \sigma \in \Sigma \) of type \( (\bigcup_c Val^{c}) \rightarrow \Sigma_{\text{inst}} \) stores for each currently existing object, i.e., an object belonging to the domain of \( \sigma \), its instance state. The set of existing objects of type \( c \) in a state \( \sigma \) is given by \( Val^{c}(\sigma) \), and \( Val_{\text{null}}^{c}(\sigma) = Val^{c}(\sigma) \cup \{null^{c}\} \). For the remaining types, \( Val^{l}(\sigma) \) and \( Val_{\text{null}}^{l}(\sigma) \) are defined correspondingly. We refer to the set \( \bigcup_l Val^{l}(\sigma) \) by \( Val^{l}(\sigma) \); \( Val_{\text{null}}^{l}(\sigma) \) denotes \( \bigcup_l Val_{\text{null}}^{l}(\sigma) \). The instance state of an object \( \alpha \in Val^{c}(\sigma) \) is given by \( \sigma(\alpha) \) with the invariant property \( \sigma(\alpha)(\text{this}) = \alpha \). We require that, given a global state, no instance variable in any of the existing objects refers to a non-existing object, i.e., \( \sigma(\alpha)(x) \in Val_{\text{null}}^{l}(\sigma) \) for all variables \( x \) from the domain of \( \tau \); 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 \( (\alpha, \tau, stm) \in T \) within one of the execution stacks of \( T \).

The semantic function \( \llbracket \cdot \rrbracket_{\text{G}}^{\neg} : (\Sigma_{\text{inst}} \times \Sigma_{\text{loc}}) \rightarrow (\text{Exp} \rightarrow Val_{\text{null}}) \) evaluates in the context of an instance local state \( (\sigma_{\text{inst}}, \tau) \) expressions containing variables from \( \text{dom}(\sigma_{\text{inst}}) \cup \text{dom}(\tau) \): Instance variables \( x \) and local variables \( u \) are evaluated to \( \sigma_{\text{inst}}(x) \) and \( \tau(u) \), respectively; this evaluates to \( \sigma_{\text{inst}}(\text{this}) \), and null has the null-reference as value, where compound expressions are evaluated by homomorphic lifting (see Table 2).

We denote by \( \tau[u \mapsto v] \) the local state which assigns the value \( v \) to \( u \) and agrees with \( \tau \) on the values of all other variables; \( \sigma_{\text{inst}}[x \mapsto v] \) is defined analogously, where \( \sigma[x \mapsto v] \) results from \( \sigma \) by assigning \( v \) to the instance variable \( x \) of object \( \alpha \). 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; \( \sigma_{\text{inst}}[\vec{y} \mapsto \vec{v}] \) and \( \sigma[\alpha, \vec{y} \mapsto \vec{v}] \) are analogous. Finally for global states, \( \sigma[\alpha \mapsto \sigma_{\text{inst}}] \) equals \( \sigma \) except on \( \alpha \); note that in case \( \alpha \notin Val^{l}(\sigma) \), the operation extends the set of existing objects by \( \alpha \), which has its instance state initialized to \( \sigma_{\text{inst}} \).

\[
\begin{align*}
[x]_{\text{G}}^{\neg_{\text{inst}}, \tau} &= \sigma_{\text{inst}}(x) \\
[u]_{\text{G}}^{\neg_{\text{inst}}, \tau} &= \tau(u) \\
[\text{this}]_{\text{G}}^{\neg_{\text{inst}}, \tau} &= \sigma_{\text{inst}}(\text{this}) \\
[\text{null}]_{\text{G}}^{\neg_{\text{inst}}, \tau} &= \text{null} \\
[f(e_1, \ldots, e_n)]_{\text{G}}^{\neg_{\text{inst}}, \tau} &= f([e_1]_{\text{G}}^{\neg_{\text{inst}}, \tau}, \ldots, [e_n]_{\text{G}}^{\neg_{\text{inst}}, \tau})
\end{align*}
\]

Table 2. Semantics of program expressions
The operational semantics of $\text{Java}_{seq}$ is given inductively by the rules of Table 3 as transitions between global configurations. The rules are formulated such a way that we can re-use 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. The remaining sequential constructs —sequential composition, conditional statement, and iteration— are standard and elided.

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

We call a configuration $\langle T, \sigma \rangle$ of a program reachable iff there is a computation $\langle T_0, \sigma_0 \rangle \rightarrow^* \langle T, \sigma \rangle$ such that $\langle T_0, \sigma_0 \rangle$ is the initial configuration of the program and $\rightarrow^*$ the reflexive transitive closure of $\rightarrow$. A local configuration $(\alpha, \tau, \text{stm}) \in T$ 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 $\text{stm}$ in the local state $\tau$ and object $\alpha$.

Assignments to instance or local variables update the corresponding state component, i.e., either the instance state or the local state (rules $\text{ASS}_{\text{inst}}$ and $\text{ASS}_{\text{loc}}$). Object creation by $u := \text{new}^c$, as shown in rule $\text{NEW}$, 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 $\text{CALL}$). After initializing the local state and passing the parameters, the thread begins to execute the method body. When returning from a method call (rule $\text{RETURN}$), the callee evaluates its return expression and passes it to the caller which subsequently updates its local state. The
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.

To be able to argue about communication histories, represented as lists of objects, we add the type Object as the supertype of all classes into the assertion language. 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 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.

Local expressions $exp_l \in LExp$ are expressions of the programming language possibly containing logical variables. The set of local expressions of type $t$ is denoted by $LExp^t$. In abuse of notation, we use $e, e', \ldots$ not only for program expressions of Table 1, but also for typical elements of local expressions. Local assertions $ass_l \in LAss$, with typical elements $p, p', q, \ldots$, 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., $z$ is required to be of type Int, Bool, or compound types 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 $\exists z \subseteq e. p$ 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 $exp_g \in GExp$, with typical elements $E, E', \ldots$, 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 $ass_g \in GAss$, with typical elements $P, Q, \ldots$, 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, i.e., the set of values $\text{Val}_{\text{null}}(\sigma)$ in a global configuration $\langle T, \sigma \rangle$.

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

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 $\omega \in \Omega$, a partial function of type $\text{LVar} \rightarrow \text{Val}_{\text{null}}$, assigning values to logical variables. We denote by $\omega[z_i \mapsto v_i]$ the logical environment that assigns the values $v_i$ to the variables $z_i$ and agrees with $\omega$ on all other variables. Similarly to local and instance state updates, the occurrence of instance and local variables in $\text{null}$ existing values and $\text{null}$.

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 $\omega \in \Omega$, a partial function of type $\text{LVar} \rightarrow \text{Val}_{\text{null}}$, assigning values to logical variables. We denote by $\omega[z_i \mapsto v_i]$ the logical environment that assigns the values $v_i$ to the variables $z_i$ and agrees with $\omega$ on all other variables. Similarly to local and instance state updates, the occurrence of instance and local variables in $\text{null}$ existing values and $\text{null}$.

Table 4. Syntax of assertions
2.3 The assertion language

\[
\begin{align*}
[z]_{\omega,\sigma}^{\omega,\sigma_{out}} & = \omega(z) \\
[x]_{\omega,\sigma}^{\omega,\sigma_{out}} & = \sigma_{inst}(x) \\
[u]_{\omega,\sigma}^{\omega,\sigma_{out}} & = \tau(u) \\
[\text{this}]_{\omega,\sigma}^{\omega,\sigma_{out}} & = \sigma_{inst}(\text{this}) \\
[\text{null}]_{\omega,\sigma}^{\omega,\sigma_{out}} & = \text{null} \\
[f(e_1, \ldots, e_n)]_{\omega,\sigma}^{\omega,\sigma_{out}} & = f([e_1]_{\omega,\sigma}^{\omega,\sigma_{out}}, \ldots, [e_n]_{\omega,\sigma}^{\omega,\sigma_{out}}) \\
([\neg p]_{\omega,\sigma}^{\omega,\sigma_{out}} & = \text{true}) & \text{iff} & ([p]_{\omega,\sigma}^{\omega,\sigma_{out}} = \text{false}) \\
([p_1 \land p_2]_{\omega,\sigma}^{\omega,\sigma_{out}}) & = \text{true}) & \text{iff} & ([p_1]_{\omega,\sigma}^{\omega,\sigma_{out}} = \text{true} \text{ and } [p_2]_{\omega,\sigma}^{\omega,\sigma_{out}} = \text{true}) \\
([\exists z.p]_{\omega,\sigma}^{\omega,\sigma_{out}}) & = \text{true}) & \text{iff} & ([\exists z.e.p]_{\omega,\sigma}^{\omega,\sigma_{out}}) = \text{true} \text{ for some } v \in \text{Val}_{null}(\sigma) \\
([\exists z \in e.p]_{\omega,\sigma}^{\omega,\sigma_{out}}) & = \text{true}) & \text{iff} & ([\exists z \in e.p]_{\omega,\sigma}^{\omega,\sigma_{out}}) = \text{true} \text{ for some } v \in \text{Val}_{null}(\sigma)
\end{align*}
\]

Table 5. Local evaluation

\[
\begin{align*}
[z]_{\sigma}^{\sigma} & = \omega(z) \\
[\text{null}]_{\sigma}^{\sigma} & = \text{null} \\
[f(e_1, \ldots, e_n)]_{\sigma}^{\sigma} & = f([E_1]_{\sigma}^{\sigma}, \ldots, [E_n]_{\sigma}^{\sigma}) \\
[E.x]_{\sigma}^{\sigma} & = \sigma([E]_{\sigma}^{\sigma})(x) \\
([\neg P]_{\sigma}^{\sigma} & = \text{true}) & \text{iff} & ([P]_{\sigma}^{\sigma} = \text{false}) \\
([P_1 \land P_2]_{\sigma}^{\sigma}) & = \text{true}) & \text{iff} & ([P_1]_{\sigma}^{\sigma} = \text{true} \text{ and } [P_2]_{\sigma}^{\sigma} = \text{true}) \\
([\exists z. P]_{\sigma}^{\sigma} & = \text{true}) & \text{iff} & ([\exists z \in v. P]_{\sigma}^{\sigma} = \text{true} \text{ for some } v \in \text{Val}_{null}(\sigma))
\end{align*}
\]

Table 6. Global evaluation

ables, the global assertional semantics does not refer to instance local states but to global states. The semantic function \([.]_{\omega}^{\sigma}\) of type \((\Omega \times \Sigma) \rightarrow (GExp \cup GAss \rightarrow \text{Val}_{null})\), shown in Table 6, gives meaning to global expressions and assertions in the context of a global state \(\omega\) and a logical environment \(\omega\). To be well-defined, \(\omega\) is required to refer only to values existing in \(\sigma\), and the expression respectively assert may only contain free variables\(^3\) from the domain of \(\omega\). 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 \(\sigma\). 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 \(z \in LVar\) evaluates to true in the context of \(\omega\) and \(\sigma\) if and only if \(P\) evaluates to true in the context of \(\omega[z \mapsto v]\) and \(\sigma\), for some value \(v \in \text{Val}_{null}(\sigma)\). Note that quantification over objects ranges over the set of existing objects and \(\text{null}\), only.

\(^3\)In global expressions \(E.x\) we treat \(x\) as a bound variable.
For a global state $\sigma$ and a logical environment $\omega$ referring only to values existing in $\sigma$ we write $\omega, \sigma \models G P$ when $P$ is true in the context of $\omega$ and $\sigma$. We write $\models G P$ if $P$ holds for arbitrary global states $\sigma$ and logical environments $\omega$ referring only to values existing in $\sigma$.

To express a local property $p$ in the global assertion language, we define the substitution $p[z/this]$ 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 $p[z/this]$ as logical variables. Formally, these local variables are replaced by fresh logical variables. We write $P(z)$ for $p[z/this]$, and similarly for expressions. For unrestricted quantifications $(\exists z'. p)[z/this]$ 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:

$$
\begin{align*}
\text{this}[z/this] &= z \\
x[z/this] &= z.x \\
u[z/this] &= u \\
(\exists z'. p)[z/this] &= \exists z'. p[z/this] \\
(\exists z' \in e. p)[z/this] &= \exists z'. (z' \in e[z/this] \land p[z/this]) \\
(\exists z' \sqsubseteq e. p)[z/this] &= \exists z'. (z' \sqsubseteq e[z/this] \land p[z/this]),
\end{align*}
$$

where $z$ is fresh.

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 2.1. (Lifting substitution)**

Let $\sigma$ be a global state, $\omega$ and $\tau$ a logical environment and local state, both referring only to values existing in $\sigma$. 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

$$
\omega, \sigma \models G p[z/this] \iff \omega, \sigma(\omega(z)), \tau \models L p.
$$

### 2.4. The proof system

The proof system has to accommodate for dynamic object creation, aliasing, method invocation, and recursion. The following section defines how to augment and annotate programs resulting in proof outlines, before Section 2.4.2 describes the proof method.

For technical convenience, we formulate verification conditions as standard Hoare-triples. The statements of these Hoare-triples may also contain assignments involving qualified references as given by the global assertion language. For the formal semantics and for re-formulated verification conditions using substitutions see [1].
2.4 The proof system

2.4.1 Proof outlines

For a 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.

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 they observe in an atomic manner. For object creation this is syntactically represented by the augmentation $u := \text{new } \vec{y} := \vec{e}$ which attaches the observation to the object creation statement. Observations $\vec{y}_1 := \vec{e}_1$ of a method call and observations $\vec{y}_3 := \vec{e}_3$ of the corresponding reception of a return value are denoted by $u := e_0.m(\vec{e}) \langle \vec{y}_1 := \vec{e}_1 \rangle^\text{call} \langle \vec{y}_3 := \vec{e}_3 \rangle^\text{return}$. The augmentation $\langle \vec{y}_2 := \vec{e}_2 \rangle^\text{call} \text{stm}; \text{return } \vec{e}_4$ 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 $\vec{y} := \vec{e} \langle \vec{y} := \vec{e} \rangle^\text{ass}$. 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 $e_1.m(\vec{e}) \langle \text{stm} \rangle$ uniquely specifies $\text{stm}$ as an alone-standing observation following an unobserved method call; using the same augmentation syntax $\langle \text{stm} \rangle$ for all kinds of observations, we would have to write $e_0.m(\vec{e}) \langle \text{stm} \rangle$ 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. 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. Points between a statement and its observation are no control points, since they are executed in a single computation step; we call them auxiliary points.

To exclude the possibility, that two multiple assignments get executed in a single computation step in 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. Formally, in each observation of a method invocation statement $e_0.m(\vec{e})$, assignments to instance variables must have the form $x := \text{if } e_0 = \text{this then } x \text{ else } e_1$.

In the following we call assignment statements with their observations, unobserved assignments, alone-standing observations, or observations of communication or object creation general as multiple assignments, since they are executed simultaneously.

For completeness, it is necessary to be able to identify objects and instances of method executions, i.e., local configurations. We identify a local configuration by the object in which it executes together with the value of its built-in auxiliary local variable conf storing a unique object-internal identifier. Its uniqueness is assured by the auxiliary instance variable counter, incremented for each new local con-
The notion of class invariant commonly used for sequential object-oriented languages differs from our notion: In a sequential setting, it would be sufficient that the class invariant holds initially and is preserved by whole method calls, but not necessarily in between.

Syntactically, each method declaration \( m(\vec{u}, \text{callee})\{\text{stm}; \text{return } e_{\text{ret}}\} \) gets extended by the built-in augmentation to \( m(\vec{u}, \text{callee})\{(\text{conf}, \text{counter} := \text{counter}, \text{counter} + 1)\text{call } \text{stm}; \text{return } e_{\text{ret}}\} \). Correspondingly for method calls \( u := e_0.m(\vec{e}) \), the actual parameter lists get extended to \( u := e_0.m(\vec{e}, (\text{this, conf})) \). 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.

**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 triple notation \( \{p\} \text{stm} \{q\} \) and write \( \text{pre}(\text{stm}) \) and \( \text{post}(\text{stm}) \) to refer to the pre- and the post-condition of a statement. For assertions at auxiliary points we use the following notation: The annotation

\[
\{p_0\} \ u := \text{new} \vec{x} \ \{p_1\}^\text{new} \ (\vec{y} := \vec{e})^\text{new} \ \{p_2\}
\]

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

\[
\{p_0\} \ u := e_0.m(\vec{e}) \ \{p_1\}^\text{call} \ (\vec{y}_1 := \vec{e}_1)^\text{call} \ \{p_2\}^\text{wait} \ \{p_3\}^\text{return} \ (\vec{y}_4 := \vec{e}_4)^\text{return} \ \{p_4\}
\]

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

\[
\{p_0\}^\text{call} \ (\vec{y}_2 := \vec{e}_2)^\text{call} \ \{p_1\} \ \text{stm}; \ \{p_2\} \ 	ext{return } e_{\text{ret}} \ \{p_3\}^\text{return} \ (\vec{y}_3 := \vec{e}_3)^\text{return} \ \{p_4\}
\]

The callee postcondition of the method call is \( p_1 \); the callee pre- and postconditions of 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.\(^4\) We require that for each method of a class, the class invariant is the precondition of the method body.

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.

\(^4\)The notion of class invariant commonly used for sequential object-oriented languages differs from our notion: In a sequential setting, it would be sufficient that the class invariant holds initially and is preserved by whole method calls, but not necessarily in between.
2.4 The proof system

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.

Before specifying the verification conditions, we first list some notation. Let $\text{Init}$ be a syntactical operator with interpretation $\text{Init}$ (see page 6). Given $\text{IVar}_c$ as the set of instance variables of class $c$ without the self-reference, and $z$ a logical variable of type $c$, let $\text{InitState}(z)$ be the global assertion $z \neq \text{null} \land \bigwedge_{x \in \text{IVar}_c} z.x = \text{Init}(x)$, expressing that the object denoted by $z$ is in its initial instance state.

Finally, 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 $u'$, and correspondingly for expressions.

**Initial correctness** A proof outline 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.

**Definition 2.1. (Initial correctness)**
Let the body of the run-method of the main class $c$ be $\{p_2\}^{\text{call}} \langle \vec{y}_2 := \vec{e}_2 \rangle^{\text{call}} \{p_3\} \text{stm}; \text{return with local variables } \vec{v} \text{ without the formal parameters, } z \in \text{LVar}^c, \text{ and } z' \in \text{LVar}^{\text{Object}}. \text{ A proof outline is initially correct, if }

$$
\begin{align*}
|_{G} & \text{Init}(z) \land \forall z'. z' = \text{null} \lor z = z' \quad \vec{v}, \text{caller} := \text{Init}(\vec{v}), (\text{null}, 0) \quad \{P_2(z)\} \quad (1) \\
|_{G} & \text{Init}(z) \land \forall z'. z' = \text{null} \lor z = z' \quad \vec{v}, \text{caller} := \text{Init}(\vec{v}), (\text{null}, 0) \quad z.\vec{y}_2 := \vec{E}_2(z) \quad \{GI \land P_3(z) \land I_c(z)\} \quad (2)
\end{align*}
$$

The assertion $\text{Init}(z) \land \forall z'. z' = \text{null} \lor z = z'$ 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 $\vec{v}, \text{caller} := \text{Init}(\vec{v}), (\text{null}, 0)$. The observation $\vec{y}_2 := \vec{E}_2(z)$ at the beginning of the run-method of the initial object $z$ is represented by the assignment $z.\vec{y}_2 := \vec{E}_2(z)$.

**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 [13] for standard sequential constructs hold. For example, the precondition of an assignment must imply its postcondition after its execution. The following condition should hold for all multiple assignments being an assignment statement with its observation, an unobserved assignment, or an alone-standing observation:
Definition 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 object creation or communication,
\[
| = \mathcal{L} \{ p_1 \} \vec{y} := \vec{e} \{ p_2 \} .
\]
(3)
The conditions for loops and conditional statements are similar. 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.

The interference freedom test  Invariance of local assertions under computation steps in which they are not involved is assured by the proof obligations of the **interference freedom test**. Its definition covers also invariance of the class invariants. Since Java_seq does not support qualified references to instance variables, we only have to deal with invariance under execution within the same object. Affecting only local variables, communication and object creation do not change the instance states of the executing objects. Thus we only have to cover invariance of assertions at control points over assignments, including observations of communication and object creation. To distinguish local variables of the different local configurations, we rename those of the assertion.

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, need not be considered and is excluded by requiring \( \text{conf} \neq \text{conf}' \).

Interference with the matching return statement in a self-communication need also not be considered, because communicating partners execute simultaneously. Let \( \text{caller}_\text{obj} \) be the first and \( \text{caller}_\text{conf} \) the second component of caller. We define \( \text{waits}_\text{for}_\text{ret}(q, \vec{y} := \vec{e}) \) by

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

The interference freedom test can now be formulated as follows:

Definition 2.3. (Interference freedom)
A proof outline is **interference free**, if for all classes \( c \) and multiple assignments \( \vec{y} := \vec{e} \) with precondition \( p \) in \( c \),
\[
| = \mathcal{L} \{ p \wedge I_c \} \vec{y} := \vec{e} \{ I_c \} .
\]
(4)

Furthermore, for all assertions \( q \) at control points in \( c \),
\[
| = \mathcal{L} \{ p \wedge q' \wedge \text{waits}_\text{for}_\text{ret}(q, \vec{y} := \vec{e}) \} \vec{y} := \vec{e} \{ q' \} .
\]
(5)
Note that if we would allow qualified references in program expressions, we would have to show interference freedom of 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.

**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 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 are allowed to 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 as follows: After communication, i.e., after creating and initializing the callee local configuration and passing on the actual parameters, first the caller, and then the callee execute their corresponding observations, all in a single computation step. Correspondingly for return, after communicating the result value, first the callee and then the caller observation gets executed. 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 \( P(z) \) for \( p[z/\text{this}] \), \( Q'(z') \) for \( q'[z'/\text{this}] \), and similarly for expressions. To avoid name clashes between local variables of the partners, we rename those of the callee.

Let \( z \) and \( z' \) be logical variables representing the caller, respectively the callee object in a method call. We assume the global invariant 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 in the assertion \( \text{comm} \), which depends on the type of communication: For method invocation \( e_0.m(\vec{e}) \), the assertion \( E_0(z) = z' \) states, that \( z' \) is indeed the callee object. Remember that method invocation hands over the return address, 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, \( \text{comm} \) additionally states \( \vec{u}' = E'(z) \), where \( \vec{u} \) and \( \vec{e} \) are the formal and the actual parameters. Returning from the run-method terminates the executing thread, which does not have communication effects.

As in the previous conditions, state changes are represented by assignments. For the example of method invocation, communication is represented by the assignment \( \vec{u}' := E'(z) \), where initialization of the remaining local variables \( \vec{v} \) is covered by \( \vec{v}' := \text{Init}(\vec{v}) \). The assignments \( z.\vec{y}_1 := E'_1(z) \) and \( z'.\vec{y}_2 := E'_2(z') \) 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. (Cooperation test: Communication)

A proof outline satisfies the cooperation test for communication, if

\[
\models \mathcal{G} \quad \{ GI \land P_1(z) \land \mathcal{Q}_1(z') \land \mathcal{C} \land z \neq \text{null} \land z' \neq \text{null} \}
\]

\[
f_{\mathcal{C}}
\]

\[
\{ P_2(z) \land \mathcal{Q}_2(z') \}
\]

\[
\models \mathcal{G} \quad \{ GI \land P_1(z) \land \mathcal{Q}_1(z') \land \mathcal{C} \land z \neq \text{null} \land z' \neq \text{null} \}
\]

\[
f_{\mathcal{C}} \cdot f_{\text{obs}1} \cdot f_{\text{obs}2}
\]

\[
\{ GI \land P_3(z) \land \mathcal{Q}_3(z') \}
\]

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

1. **CALL**: For all statements \(\{p_1\}_{u_{\text{ret}}} := e_0.m(\vec{e}) \quad \{p_2\}_{\text{call}} \langle \vec{y}_1 := \vec{e}_1 \rangle_{\text{call}} \quad \{p_3\}_{\text{wait}}\) (or such without receiving a value) in class \(c\) with \(e_0\) of type \(c'\), where method \(m\) of \(c'\) has body \(\{q_2\}_{\text{call}} \langle \vec{y}_2 := \vec{e}_2 \rangle_{\text{call}} \quad \{q_3\}_{\text{stm}}; \text{return}_{e_{\text{ret}}}\), formal parameters \(\vec{u}\), and local variables \(\vec{v}\) except the formal parameters. The callee class invariant is \(q_1 = L_c\). The assertion \(\mathcal{C}\) is given by \(E_0(z) = z'.\) Furthermore, \(f_{\mathcal{C}}\) is \(u'\), \(v' := E(z), \text{Init}(v), f_{\text{obs}1}\) is \(z; y_1 := E_1(z), \) and \(f_{\text{obs}2}\) is \(z'; y_2 := E_2(z').\)

2. **RETURN**: For all \(u_{\text{ret}} := e_0.m(\vec{e}) \quad \{p_1\}_{\text{call}} \quad \{p_2\}_{\text{wait}} \quad \{p_3\}_{\text{ret}}\) (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 \(\{q_1\}_{\text{run}}\) \(\text{return}_{e_{\text{ret}}} \quad \{q_2\}_{\text{ret}} \quad \{q_3\}_{\text{ret}}\), and formal parameter list \(\vec{u}\), the above equations must hold with \(\mathcal{C}\) given by \(E_0(z) = z' \land u' = E(z), \) and where \(f_{\mathcal{C}}\) is \(u_{\text{ret}} := E'_{\text{ret}}(z'), \text{obs}1\) is \(z'; y_3 := E_3(z'), \) and \(f_{\text{obs}2}\) is \(z; y_4 := E_4(z).\)

3. **RETURN_run**: For \(\{q_1\}_{\text{run}}\) \(\text{return}_{q_2} \quad \{q_3\}_{\text{ret}}\) occurring in the run-method of the main class, \(p_1 = p_2 = p_3 = \text{true}, \mathcal{C} = \text{true}, \) and furthermore \(f_{\mathcal{C}}\) and \(f_{\text{obs}2}\) are the empty statement, and \(f_{\text{obs}1}\) is \(z'; y_3 := E_3(z').\)

Besides method calls and returns, 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 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 \(\text{LVar}^\text{listObject}\) 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 \(\text{Fresh}(z', u)\) defined as \(\text{InitState}(u) \land u \notin z' \land \forall v. \; v \in z' \lor v = u\) (see page 15 for the definition of \(\text{InitState}\)). 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 existential quantification in the assertion to range over objects from \(z',\) only. So let \(P\) be a global assertion and \(z' \in \text{LVar}^\text{listObject}\) a logical variable not occurring in \(P\). Then \(P \downarrow z'\) is the global assertion \(P\) with all quantifications \(\exists z.\; P'\) replaced by \(\exists z. \; \text{obs}(z) \subseteq z' \land P'\), where \(\text{obs}(v)\) denotes the set of objects occurring in the value \(v\). The following lemma formulates the basic property of the projection operator:
3.1 Syntax

<table>
<thead>
<tr>
<th></th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>meth ::= m(u, . . ., u) { stm; return exp; ret}</code></td>
<td></td>
</tr>
<tr>
<td><code>meth_run ::= run() { stm; return }</code></td>
<td></td>
</tr>
<tr>
<td><code>class ::= class c{meth...meth meth_run meth_start}</code></td>
<td></td>
</tr>
<tr>
<td><code>class_main ::= class</code></td>
<td></td>
</tr>
<tr>
<td><code>prog ::= (class...class_classmain)</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 7. Java conc abstract syntax

Lemma 2.2. Assume a global state $\sigma$, an extension $\sigma' = \sigma[\alpha \mapsto \sigma_{\text{inst}}]$ for some $\alpha \in \text{Val}^c$, $\alpha \notin \text{Val}(\sigma)$, and a logical environment $\omega$ referring only to values existing in $\sigma$. Let $v$ be the sequence consisting of all elements of $\bigcup_c \text{Val}^{c}_{\text{null}}(\sigma)$. Then for all global assertions $P$ and logical variables $z' \in \text{LVar}^{\text{list Object}}$ not occurring in $P$,

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

Thus a predicate $(\exists u. P) \downarrow z'$, evaluated immediately after the instantiation, 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 2.5. (Cooperation test: Instantiation)

A proof outline satisfies the cooperation test for object creation, if for all classes $c'$ and statements

$$\{p_1\} u := \text{new}^c \{p_2\}^\text{new} \langle \vec{y} := \vec{E}(z) \rangle \{P_3(z)\}$$

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

3. The concurrent language

In this section we extend the language $\text{Java}_\text{seq}$ to a concurrent language $\text{Java}_\text{conc}$ by allowing dynamic thread creation. Again, we define syntax and semantics of the language, before formalizing the proof system for the concurrent language.

3.1 Syntax

Expressions and statements can be constructed as in $\text{Java}_\text{seq}$. The abstract syntax of the remaining constructs is summarized in Table 7. As we focus on concurrency aspects, all classes are Thread classes in the sense of Java: Each class contains a pre-defined 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 8. The invocation of a start-method brings a new thread into being (rule $CALL_{start}$). Only the first invocation of the start-method has this effect (rule $CALL_{skip_{start}}$).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CALL_{start}$</td>
<td>$\beta = [e]^{\alpha, \tau}<em>{T} \in Val(\sigma) \quad \neg started(T \cup {\xi (\alpha, \tau, e.start(); stm)}, \beta)$ $\quad (T \cup {\xi (\alpha, \tau, e.start(); stm)}, \sigma) \rightarrow (T \cup {\xi (\alpha, \tau, stm), (\beta, \sigma</em>{\tau, e.body}; \text{run}_{\tau, e})}, \sigma)$</td>
</tr>
<tr>
<td>$CALL_{skip_{start}}$</td>
<td>$\beta = [e]^{\alpha, \tau}<em>{T} \in Val(\sigma) \quad started(T \cup {\xi (\alpha, \tau, e.start(); stm)}, \beta)$ $\quad (T \cup {\xi (\alpha, \tau, e.start(); stm)}, \sigma) \rightarrow (T \cup {\xi (\alpha, \tau, stm), (\beta, \sigma</em>{\tau, \text{run}_{\tau, e}})}, \sigma)$</td>
</tr>
</tbody>
</table>

Table 8. $Java_{conc}$ operational semantics

### 3.3. The proof system

In contrast to the sequential language, the proof system additionally has to accommodate for dynamic thread creation and shared-variable concurrency. Before describing the proof method, we show how to extend the built-in augmentation of the sequential language.

#### 3.3.1. Proof outlines

To get 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 an auxiliary formal parameter thread. Additionally, we extend the auxiliary formal parameter caller by the caller thread identity, i.e., let caller be of type $Object \times \text{Int} \times \text{Thread}$, storing the identities of the caller object, the

---

5In Java an exception is thrown if the thread is already started but not yet terminated.

6The worked-off local configuration $(\alpha, \tau, e)$ is kept in the global configuration to ensure that the thread of $\alpha$ cannot be started twice.
calling local configuration, and the caller thread. Note that the thread identities of caller and callee are the same in all cases but the invocation of a start-method. The run-method of the initial object is executed with the parameters (thread, caller) having the values \((e_0, (null, 0, null))\), where \(e_0\) is the initial object. The boolean instance variable 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 \((\vec{u}, \text{thread}, \text{caller})\). Correspondingly for the caller, each actual parameter list \(\vec{e}\) in statements invoking a method different from start gets extended to \((\vec{e}, \text{thread}, (\text{this}, \text{conf}, \text{thread}))\). The invocation of the parameterless start-method of an object \(e_0\) gets the actual parameter list \((e_0, (\text{this}, \text{conf}, \text{thread}))\). Finally, the callee observation at the beginning of the run-method executes started := true. The variables conf and counter are updated as in the previous section.

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 \(x := e_0 = \text{this} \land \neg\text{started} \text{ then } x \text{ else } \text{fi}\).

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 \(\{p_1\} e_0.\text{start}(\vec{e}) \{p_2\}^{\text{call}} (stm)^{\text{call}} \{p_3\}\).

### 3.3.2 Verification conditions

Initial correctness changes only, in that the formal parameters thread and caller get the initial values \(z\) and \((null, 0, null)\). 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. Again, to distinguish local variables of the different local configurations, we rename those of the assertion which we show to be 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. If \(q\) and \(\vec{y} := \vec{e}\) belong to the same thread, i.e., \(\text{thread} = \text{thread}'\), 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 any case except for the self-invocation of the start-method: The precondition of such a method invocation cannot interfere with the corresponding observation of the callee. To describe this setting, we define \(\text{self.start}(q, \vec{y} := \vec{e})\) by caller = (this, conf', thread') iff \(q\) is the precondition of a method invocation \(e_0.\text{start}(\vec{e})\) and the assignment is the callee observation at the beginning of the run-method, and by false otherwise.

**Definition 3.1. (Interference freedom)**

A proof outline is interference free, if the conditions of Definition 2.3 hold with \(\text{waits}\text{.for}\text{.ret}(q, \vec{y} := \vec{e})\) replaced by

\[
\text{interleavable}(q, \vec{y} := \vec{e}) \overset{\text{def}}{=} \text{thread} = \text{thread}' \rightarrow \text{waits}\text{.for}\text{.ret}(q, \vec{y} := \vec{e}) \land \text{thread} \neq \text{thread}' \rightarrow \neg\text{self.start}(q, \vec{y} := \vec{e}) .
\]
The cooperation test. The cooperation test for object creation is not influenced by adding concurrency, but we have to extend the cooperation test for communication by defining additional conditions for thread creation. Invoking the start-method of an object whose thread is already started does not have communication effects. 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. Note that this condition applies now to all threads.

Definition 3.2. (Cooperation test: Communication)
A proof outline satisfies the cooperation test for communication, if the conditions of Definition 2.4 hold for the statements listed there with \( m \neq \text{start} \), and additionally in the following cases:

1. CALL\text{start}: For all statements \( \{p_1\} e_0.\text{start}(\vec{e}) \{p_2\}\text{call} \langle \vec{y}_1 := \vec{e}_1 \rangle \text{call} \{p_3\} \) in class \( c \) with \( e_0 \) of type \( c' \), comm is given by \( E_0(z) = z' \wedge \neg z'.\text{started} \), where \( \{q_2\}\text{call} \langle \vec{y}_2 := \vec{e}_2 \rangle \text{call} \{q_3\} \) \( \text{stm; return} \) 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, \( f_{\text{comm}} \) is \( \vec{u}', \vec{v}' := E(z), \text{Init}(\vec{v}) \), \( f_{\text{obs1}} \) is \( z.\vec{y}_1 := E_1(z) \), and \( f_{\text{obs2}} \) is \( z'.\vec{y}'_2 := E_2'(z') \).

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

4. Reentrant monitors
In this section we extend the concurrent language with monitor synchronization. Again, we define syntax and semantics of the language \( \text{Java}_{\text{synch}} \), before formalizing the proof system.

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 which 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, which 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 efficient 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 re-apply 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 9.

Methods get decorated by a modifier \( \text{modif} \) distinguishing between non-synchronized and synchronized methods.\(^7\) In the sequel we also refer to statements in the body of a synchronized method as being

\(^7\)Java does not have the “non-synchronized” modifier; methods are non-synchronized by default.
4.2 Semantics

The operational semantics extends the semantics of Java_{conc} by the rules of Table 10, where the \texttt{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 \texttt{owns}, defined below.

The remaining rules handle the semantics of the monitor methods \texttt{wait}, \texttt{notify}, and \texttt{notifyAll}. In all three cases the caller must own the lock of the callee object (rule \texttt{CALL-monitor}). A thread can block itself on an object whose lock it owns by invoking the object’s \texttt{wait}-method, thereby relinquishing the lock and placing itself into the object’s wait set. Formally, the wait set \texttt{wait}(T, \alpha) of an object is given as the set of all stacks in T with a top element of the form \((\alpha, \tau, \texttt{?signal}; \texttt{stm})\). After having put itself on ice, the thread awaits notification by another thread which invokes the \texttt{notify}-method of the object. The \texttt{!signal} statement in the \texttt{notify}-method thus reactivates a non-deterministically chosen single thread waiting for notification on the given object (rule \texttt{SIGNAL}). Analogously to the wait set, the notified set \texttt{notified}(T, \alpha) of \alpha is the set of all stacks in T with top element of the form \((\alpha, \tau, \texttt{return getlock})\), i.e., threads which have been notified and are trying to get hold of the lock again. According to rule \texttt{RETURNwait}, the receiver can continue after notification in executing \texttt{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 \texttt{signal-and-continue} monitor discipline [12]. If no threads are waiting on the object, the \texttt{!signal} of the notifier is without effect (rule \texttt{SIGNAL-skip}). The \texttt{notifyAll}-method generalizes notify in that all waiting threads are notified via the \texttt{!signalAll}-broadcast (rule \texttt{SIGNAL_ALL}). The effect of this statement is given by defining \texttt{signal}(T, \alpha) as \(T \setminus \texttt{wait}(T, \alpha)\) \(\cup \{ \xi \circ (\beta, \tau, \texttt{stm}) \mid \xi \circ (\beta, \tau, \texttt{?signal}; \texttt{stm}) \in \texttt{wait}(T, \alpha) \}\).

Using the wait and notified sets, we can now formalize the \texttt{owns} predicate: A thread \(\xi\) owns the lock of \(\beta\) iff \(\xi\) executes some synchronized method of \(\beta\), but not its \texttt{wait}-method. Formally, \texttt{owns}(T, \beta) is synchronized. Furthermore, we consider the additional predefined methods \texttt{wait}, \texttt{notify}, and \texttt{notifyAll}, whose definitions use the auxiliary statements \texttt{!signal}, \texttt{!signalAll}, \texttt{?signal}, and \texttt{return getlock}.

\begin{table}[h]
\centering
\begin{tabular}{ll}
\texttt{modif} & ::= nsync | sync \\
\texttt{meth} & ::= \texttt{modif(m(u, \ldots, u)}{ \texttt{stm}; \texttt{return exp}_\text{ret} } \\
\texttt{meth_run} & ::= \texttt{nsync run()}{ \texttt{stm}; \texttt{return } } \\
\texttt{meth_wait} & ::= \texttt{nsync wait()}{ \texttt{?signal}; \texttt{return getlock} } \\
\texttt{meth_notify} & ::= \texttt{nsync notify()}{ \texttt{!signal}; \texttt{return } } \\
\texttt{meth_predef} & ::= \texttt{meth_start meth_wait meth_notify meth_notifyAll} \\
\texttt{class} & ::= \texttt{class c\{meth...meth meth meth_predef\}} \\
\texttt{class_main} & ::= \texttt{class} \\
\texttt{prog} & ::= \langle \texttt{class...class class_main} \rangle \\
\end{tabular}
\caption{Java_{synch} abstract syntax}
\end{table}

\texttt{Java’s Thread} class additionally support methods for suspending, resuming, and stopping a thread, but they are deprecated and thus not considered here.

\texttt{Java}’s Thread class additionally support methods for suspending, resuming, and stopping a thread, but they are deprecated and thus not considered here.
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 for synchronization, reentrant monitors, and thread coordination. First we define how to extend the augmentation of Java\textsubscript{conc}, before we describe the proof method.

#### 4.3.1. Proof outlines

To capture mutual exclusion and the monitor discipline, the 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. All auxiliary variables are initialized as usual. For values thread of type Thread and wait of type list(Thread × Int), we will also write thread ∈ wait instead of (thread, n) ∈ wait for some n. If the order of the elements of a sequence is not relevant, we apply also set theoretical operations to them.

Syntactically, besides the augmentation of the previous section, the callee observation at the beginning and at the end of each synchronized method body executes \( \text{lock} := \text{inc}(\text{lock}) \) and \( \text{lock} := \text{dec}(\text{lock}) \), respectively. The semantics of incrementing the lock \( [\text{inc}(\text{lock})]_{\text{inst}}^{\text{free}} \) is \((\tau(\text{thread}), n+1)\) for \( \sigma_{\text{inst}}(\text{lock}) = (\alpha, n) \). Decreasing \( \text{dec}(\text{lock}) \) is inverse: \([\text{dec}(\text{lock})]_{\text{inst}}^{\text{free}} \) with \( \sigma_{\text{inst}}(\text{lock}) = (\alpha, n) \) is \((\alpha, n-1)\) if \( n > 1 \), and free otherwise.

Instead of the auxiliary statements of the semantics, notification is represented in the proof system by auxiliary assignments operating on the wait and notified variables. That means, the auxiliary \( \text{?signal}, \text{!signal}, \text{!signal} \) all statements get replaced by auxiliary assignments\(^9\). Entering the wait-method gets the observation wait, lock := \text{wait} ∪ \{\text{lock}\}, free; returning from the wait-method observes lock, notified := \text{get}(\text{notified}, \text{thread}), \text{notified} \setminus \{\text{get}(\text{notified}, \text{thread})\}. \) For a thread \( \alpha \in \text{Val}^{\text{Thread}} \) and a list \( \text{notified} \in \text{Val}^{\text{list}(\text{Thread} \times \text{Int})} \), \( \text{get}(\text{notified}, \alpha) \) retrieves the value \((\alpha, n)\) from the list. The semantics assures uniqueness of the association. The \( \text{!signal} \) statement of the notify-method is represented by the auxiliary assignment wait, notified := notify(wait, 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. Finally, the \( \text{!signal} \) all statement of the notifyAll-method is represented by the auxiliary assignment notified, wait := notified ∪ wait, \( \emptyset \).

### 4.3.2. Verification conditions

Initial and local correctness agree with those for Java_{conc}. In case of notification, local correctness covers also invariance for the notifying thread, as the effect of notification is captured by an auxiliary assignment.

**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 is in the wait or notified set of the object; otherwise, both executing local configurations are in the same call chain. Thus we assume that either not both the assignment and the assertion occur in a synchronized method, or the assertion is at a control point waiting for return.\(^{10}\)

**Definition 4.1. (Interference freedom)**

A proof outline is interference free, if Definition 3.1 holds in all cases, such that either not both \( p \) and \( q \) occur in a synchronized method, or \( q \) is at a control point waiting for return.

For notification, we require also invariance of the assertions for the notified thread. 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 the 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

---

\(^9\)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.

\(^{10}\)This condition is not necessary for a minimal proof system, but reduces the number of verification conditions.
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.

5. Exception handling

The cooperation test We extend the cooperation test for Java$_{monc}$ with synchronization and 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 $z'.lock = \text{free} \lor \text{thread}(z'.lock) = \text{thread}$, where thread is the caller thread, $z'$ is the callee object, and where $\text{thread}(z'.lock)$ 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. Note that the global invariant is not affected by the object-internal monitor signaling mechanism, which is represented by auxiliary assignments.

Definition 4.2. (Cooperation test: Communication)

A proof outline satisfies the cooperation test for communication, if the conditions of Definition 3.2 hold for the statements listed there with the exception of the CALL-case, and additionally in the following cases:

1. **CALL**: For all statements $\{p_1\} u_{ret} := e_0.m(c') \{p_2\}^{\text{call}} (y_1 := e_1)^{\text{call}} \{p_3\}^{\text{wait}}$ (or such without receiving a value) in class $c$ with $e_0$ of type $c'$, where method $m \notin \{\text{start, wait, notify, notifyAll}\}$ of $c'$ is synchronized with body $\{q_2\}^{\text{call}} (y_2 := e_2)^{\text{call}} \{q_3\} \text{ stm; return } e_{ret}$, 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 $E_0(z) = z' \land (z'.lock = \text{free} \lor \text{thread}(z'.lock) = \text{thread})$. Furthermore, $f_{comm}$ is $\vec{u}', \vec{v}' := E(z), \text{Init} (\vec{v}), f_{obs1}$ is given by $z,y_1 := E_1(z)$, and $f_{obs2}$ is $z',y_2' := E_2(z')$. If $m$ is not synchronized, $z'.lock = \text{free} \lor \text{thread}(z'.lock) = \text{thread}$ in $comm$ is dropped.

2. **CALL$_{monitor}$**: For $m \in \{\text{wait, notify, notifyAll}\}$, comm is $E_0(z) = z' \land \text{thread}(z'.lock) = \text{thread}$.

3. **RETURN$_{wait}$**: For $\{q_1\} \text{return } \{q_2\}^{\text{ret}} (y_3 := e_3)^{\text{ret}} \{q_3\}$ in a wait-method, comm is $E_0(z) = z' \land \vec{u}' = E(z) \land z'.lock = \text{free} \land \text{thread}' \in z'.\text{notified}$.

5. Exception handling

In this section we extend the previous language with exception handling. Again, we define syntax and semantics of the language Java$_{exe}$, before formalizing the proof system.

5.1. Syntax

We introduce additional statements for exception throwing and handling, as shown in Table 9. The abstract syntax of the remaining constructs is as for the previous language.
5.2 Semantics

Table 11. JavaExc abstract syntax

Table 12. JavaExc operational semantics (1)

5.2 Semantics

Exceptions allow a special form of error handling: If something unexpected or unallowed happens, the executing thread may throw an exception, which is an object of an arbitrary type. The empty reference cannot be thrown. If an exception has been thrown by a thread, then the normal flow of control gets interrupted, and control tries to find the “nearest” exception handler handling exceptions of the given type, as explained below.

The operational semantics extends the semantics of Java_{synch} by the rules of the Tables 12 and 13, covering exception handling. In the semantics we use the type Object, as already introduced for augmentation and annotation, being the supertype of all classes. Note that no objects of type Object can be created, thus preserving monomorphism.

Throwing and catching exceptions are syntactically represented by throw statements and by try-catch-finally blocks. During the execution of a try-catch-finally block try \( stm_0 \) catch (\( c_1 u_1 \)) \( stm_1 \ldots \) catch (\( c_n u_n \)) \( stm_n \) finally \( stm_{n+1} \) yrt, the corresponding local configuration contains an “open” try-construct like e.g. \( stm'_0 \): catch (\( c_1 u_1 \)) \( stm_1 \ldots \) catch (\( c_n u_n \)) \( stm_n \) finally \( stm_{n+1} \) yrt (rule TRY). We

\[\tau' = \tau [\text{exc} \mapsto \tau(\text{exc}) \circ \text{null}]\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T \cup { \xi \circ (\alpha, \tau, \text{try } stm_0; \text{catch } (c_1 u_1) \text{ } stm_1 \ldots; \text{catch } (c_n u_n) \text{ } stm_n \text{ finally } stm_{n+1} \text{ yrt; } stm') }, \sigma )</td>
<td>TRY</td>
</tr>
<tr>
<td>( T \cup { \xi \circ (\alpha, \tau, \text{catch } (c_1 u_1) \text{ } stm_1 \ldots; \text{catch } (c_n u_n) \text{ } stm_n \text{ finally } stm_{n+1} \text{ yrt; } stm') }, \sigma )</td>
<td>FINALLY</td>
</tr>
<tr>
<td>( \tau(\text{exc}) = \beta_0 \circ \ldots \circ \beta_k \circ \beta_{k+1} )</td>
<td>( \tau' = \tau [\text{exc} \mapsto \beta_0 \circ \ldots \circ \beta_k] \circ \text{top} \mapsto \beta_{k+1} )</td>
</tr>
<tr>
<td>( T \cup { \xi \circ (\alpha, \tau, \text{try; } stm) }, \sigma )</td>
<td>( T \cup { \xi \circ (\alpha, \tau', \text{try; } stm') }, \sigma )</td>
</tr>
</tbody>
</table>

11 In Java only objects extending Throwable may be thrown.
12 In Java, a NullPointerException is thrown in this case.
Note that for example try-closed exc

of type which is not yet caught; a null-reference means the absence of such an exception. The additional variable

is used to store the value of an exception which should be rethrown.

call such blocks also statements, even if they are no statements in a strong syntactical sense.\(^{13}\) Statements in which no such open try blocks occur are called try-closed.

The semantics uses the local variable exc of types list Object with initial value \(\epsilon\), to store thrown but not yet caught exceptions. In nested try-catch-finally statements, each try-catch-finally statement has its own element in the sequence exc which is used to remember if there is an exception throw in that block which is not yet caught; a null-reference means the absence of such an exception. The additional variable top of type Object is used to store the value of an exception which should be rethrown.

Entering a try-catch-finally block appends a null-reference to the value of exc, expressing that there is no thrown but not yet caught exception in that block (rule TRY).

The execution of a try-catch-finally block consists of the execution of the try statement until an exception is thrown or the try statement terminates. If an exception is thrown, and if there is a corresponding catch-clause handling exceptions of the given type, then this catch-clause (rule THROW\(_1\)) and the finally clause (rule FINALLY with \(n = 0\)) get executed. Otherwise, if no exceptions have been thrown

\(^{13}\)Note that for example catch \((c_2 \ u_2) \ \text{stm}_2\) is not a statement.
(rules FINALLY) or if there is no corresponding catch clause (rule THROW2), then the finally clause gets executed. Also throwing an exception in a catch-clause (rule THROW2 with \( n = 0 \)) causes the control to move to the finally block. Throwing an exception in the finally-clause overwrites exceptions thrown in the try- or catch-clauses (rule THROW3).

Exiting a try-catch-finally block removes the last element of exc and stores it in the variable top (rule YRT). If the value of top is different from the null reference, i.e., if there was a thrown but not caught exception in the block, then the exception gets rethrown.

Throwing an exception outside try-catch-finally blocks causes the control to return to the caller, and to rethrow the exception there (rule THROW4). For run-methods, throwing such an exception terminates the executing thread (rule THROW5).

If, due to a thrown exception, control returns to the caller, and if the callee local configuration is the only one in the stack which executes a synchronized method of the callee object, then its termination gives the lock free like normal termination. This happens after evaluating the corresponding finally clause within the method, if any. Note that returning from a method due to exception handling does not hand over the return value as specified in the return statement.

### 5.3. The proof system

The proof system has to accommodate additionally for exception handling. First we define how to extend the augmentation of Java\textsubscript{synch}, before we describe the proof method.

#### 5.3.1. Proof outlines

We extend the local and the global assertion language with assertions of the form \( \text{hastype}(e,c) \) and \( \text{hastype}(E,c) \), respectively, which state that the value of \( e \) respectively \( E \) is of type \( c \); we need this construct to be able to express which type of expression has been thrown. Remember that the programming language is monomorph, and thus the association is unique.

Augmentation and annotation of the previous section get extended as follows: Exception throwing gets augmented and annotated in the form

\[
\{p_0\} \text{ throw } u \{p_1\} \begin{cases} \text{throw} \quad \{\vec{y} := \vec{e}\} \text{ throw } \{p_2\} \end{cases}.
\]

Exception throwing and its observation are executed in a single computation step, in this order. The assertion \( p_0 \) is the precondition of the throw statement. Note that the control point annotated by the postcondition \( p_2 \) is not reachable. The assertion \( p_1 \) describes the auxiliary point directly after exception throwing and before its observation \( \vec{y} := \vec{e} \).

Furthermore, we extend the augmentation and annotation of method call statements, in order to logically capture the control flow if control returns to the caller due to an exception, which gets rethrown:

\[
\{p_0\} \quad u := e_0.m(\vec{e}) \quad \{p_1\} \begin{cases} \text{call} \quad \{\vec{y}_1 := \vec{e}_1\} \text{call} \\
\{p_3\} \begin{cases} \text{ret} \quad \{\vec{y}_4 := \vec{e}_4\} \text{ret} \\
\{p_5\} \begin{cases} \text{rethrow} \quad \{\vec{y}_{\text{thr}} := \vec{e}_{\text{thr}}\} \text{rethrow}
\end{cases}
\end{cases}
\{p_6\}.
\]

Again, after control returns but before the corresponding observation the assertion \( p_3 \) should hold. If control returns due to an exception, the assertion \( p_4 \) should hold after the observation. In this case the
exception has to be rethrown; \( p_5 \) describes the state directly after rethrowing the exception in top prior to its observation \( \vec{y}_{thr} := e_{thr} \). Note that this observation does not have a postcondition, because the control point after the observation is not reachable. Note furthermore that only \( p_0, p_2, p_4, \) and \( p_6 \) annotate a control points. If control returns due to normal method termination, the assertion \( p_6 \) should hold after the observation \( \vec{y}_4 := e_4 \).

The augmentation and annotation of try-catch-finally statements is as shown in Definition 5.1. The assertion \( p \) is the precondition of the try-catch-finally block. The assertion \( p_{try} \) should hold after entering the try-block and before the corresponding observation \( \vec{y}_{try} := e_{try} \), where the assertion \( p_0 \) describes the control point after observation, and \( p'_0 \) is the postcondition of the whole try-block. The pre- and postconditions of the first and of the last catch blocks are \( p_1 \) and \( p'_1 \) respectively \( p_n \) and \( p'_n \). The finally block has the pre- and postconditions \( p_{fin} \) and \( p'_{fin} \). After exiting the finally block, \( p_{yrt} \) should hold prior to the observation \( \vec{y}_{yrt} := e_{yrt} \) of exiting. If there is an exception to be rethrown, the assertion \( p_{exc} \) is required to hold after the observation of \( e_{yrt} \), \( p_{thr} \) should hold after rethrowing and prior to its observation \( \vec{y}_{thr} := e_{thr} \). Again, this observation does not have a postcondition, because the control point after the observation is not reachable. Note that \( p_{try}, p_{yrt}, \) and \( p_{thr} \) annotate auxiliary points. If there is no exception to be rethrown, the assertion \( p' \) should hold after exiting the finally-block and executing the corresponding observation.

Remember that the local variable \( exc \) of type \( \text{list Object} \) with initial value \( \epsilon \) stores the thrown but not yet caught exceptions in nested try-catch-finally blocks. The variable \( top \) stores the value of an exception to be rethrown. We use the assertion \( \text{thrown} \) as a shortcut for \( \text{tail}(exc) \neq \text{null} \), where the function \( \text{tail}(v) \) gives the last element of the sequence \( v \). We use also the function \( \text{head}(v) \) which returns the sequence \( v \) without its last element\(^{14} \). Note that the variables \( exc \) and \( top \) are \textit{local}. In the concurrent setting, all threads have their own exception mechanism, which are independent of each other.

The augmentation for the built-in auxiliary variable \( lock \) gets extended to capture the case when a thread terminates the execution of a synchronized method due to a thrown exception: We additionally observe each \textit{throw} statement outside try-catch-finally blocks in a synchronized method by the assignment \( lock := \text{dec}(lock) \).

Since the global invariant should describe object-external behavior, we required that instance variables occurring in the global invariant may be changed by observations of communication or object creation only. Since the execution of throw statements outside try-catch-finally blocks cause the control to move to the caller, i.e., its effect is also object-external, the observations of such throw statements may also change the values of instance variables referred to in the global invariant.

### 5.3.2. Verification conditions

Initial correctness and interference freedom agree with those for Java\textsubscript{synch}. Note that exception throwing and handling do not modify instance states. Invariance under their observations, which are multiple assignments, is already included in the interference freedom test conditions of the previous section.

**Local correctness** Additionally to the local correctness conditions of the previous section, we introduce new conditions to cover the control flow of exception handling.

\[^{14}\text{These functions are applied to non-empty sequences only.}\]
5.3 The proof system

Entering a try block pushes an empty reference on the exception stack (rule \text{TRY}); thus the precondition of a try-catch-finally statement should imply the precondition of the try block after entering the block and executing the observation of the try keyword as stated in Condition (11). Furthermore, the precondition of the observation should hold directly after entering the block, prior to the observation, as formalized in Condition (10).

If no exceptions have been thrown in a try or in a catch block, then after termination of the block execution continues in the finally block (rule \text{FINALLY}); the postcondition of each try and catch block should imply the precondition of the finally block, as required by Condition (12).

Exiting the finally block (rule \text{YRT}) is covered by the Conditions (13)-(15). Condition (13) assures that \( p_{\text{yrt}} \) holds after exiting the finally block but before its observation. Remember that in case of a thrown but not yet caught exception the exception is stored in the variable top, and becomes rethrown after the block; in this case the assertion \( p_{\text{exc}} \) is required to hold after the observation of \( y_{\text{rt}} \) and prior to rethrowing, as stated in Condition (15). If no exceptions must be rethrown, Condition (14) assures that the assertion \( p' \) is satisfied after the termination of the try-catch-finally block.

If an exception has been thrown in a try block (rules \text{THROW1} and \text{THROW2}), then the precondition of the throw statement must imply the precondition of the corresponding catch block, if any, after throwing and its observation, and the precondition of the finally block otherwise; these cases are covered by the Conditions (17) and (19). Satisfaction of the preconditions of the corresponding observations is covered by the Conditions (16) and (18). The conditions for exception throwing in a catch block, in a finally block, or outside try-catch-finally blocks in run methods are modifications of the above conditions.

Remember that if an exception is thrown but not yet caught, the execution will not continue after the try-catch-finally block, but move to the next outer try-catch-finally block or to the caller configuration. The latter (rule \text{THROW4}) is covered by the conditions of the cooperation test for exception handling.

\textbf{Definition 5.1. (Local correctness: Exception handling)}

A proof outline is \textit{locally correct} under exception handling, if for all statements \( stm \) of the form

\[
\begin{align*}
\{p\} & \quad \text{try} \quad \{p_{\text{try}}\}^{\text{try}} \quad \langle \vec{y}_{\text{try}} := \vec{e}_{\text{try}} \rangle^{\text{try}} \quad \{p_0\} \quad stm_{\text{try}}: \quad \{p'_0\} \\
\text{catch}(c_1 u_1) & \quad \{p_1\} \quad stm_1: \quad \{p'_1\} \\
\ldots & \\
\text{catch}(c_n u_n) & \quad \{p_n\} \quad stm_n: \quad \{p'_n\} \\
\text{finally} & \quad \{p_{\text{fin}}\} \quad stm_{\text{fin}}: \quad \{p'_{\text{fin}}\} \\
\{p_{\text{exc}}\}^{\text{exc}} & \\
\{p'\} & \quad \text{yrt} \quad \{p_{\text{yrt}}\}^{\text{yrt}} \quad \langle \vec{y}_{\text{yrt}} := \vec{e}_{\text{yrt}} \rangle^{\text{yrt}} \quad \{p_0\} \quad stm_{\text{yrt}}: \quad \{p'_0\} \\
\{p_{\text{thr}}\}^{\text{rethrow}} & \quad \langle \vec{y}_{\text{thr}} := \vec{e}_{\text{thr}} \rangle^{\text{rethrow}} \quad \{p'\} \\
\end{align*}
\]
and for all $0 \leq i \leq n$,

$$\models \{p\} \text{exc} := \text{exc} \circ \text{null} \{p_{\text{try}}\},$$

$$\models \{p\} \text{exc} := \text{exc} \circ \text{null}; \ y_{\text{try}} := \tilde{e}_{\text{try}} \{p_{0}\},$$

$$\models p_{i} \rightarrow p_{\text{fin}},$$

$$\models \{p'_{\text{fin}}\} \text{exc}, \top := \text{head}(\text{exc}), \text{tail}(\text{exc}) \{p_{\text{retry}}\},$$

$$\models \{p'_{\text{fin}}' \land \text{tail}(\text{exc}) = \text{null}\} \text{exc}, \top := \text{head}(\text{exc}), \text{tail}(\text{exc}); \ y_{\text{retry}} := \tilde{e}_{\text{retry}} \{p'\} ,$$

$$\models \{p'_{\text{fin}}' \land \text{tail}(\text{exc}) \neq \text{null}\} \text{exc}, \top := \text{head}(\text{exc}), \text{tail}(\text{exc}); \ y_{\text{retry}} := \tilde{e}_{\text{retry}} \{p_{\text{exc}}\},$$

and for all statements \(\{q_{0}\} \text{throw } e \{q_{1}\}^{\text{throw}} \langle y := \tilde{c} \rangle^{\text{throw}}\) in \( stm_{\text{try}}\) which do not occur in an inner try-catch-finally block inside \( stm_{\text{try}}\), and for all $1 \leq i \leq n$,

$$\models \{q_{0} \land e \neq \text{null} \land \text{hastype}(e, c_{i}) \land \forall 1 \leq j < i. \neg \text{hastype}(e, c_{j})\} \quad u_{i} := e \{q_{1}\} \quad (16)$$

$$\models \{q_{0} \land e \neq \text{null} \land \text{hastype}(e, c_{i}) \land \forall 1 \leq j < i. \neg \text{hastype}(e, c_{j})\} \quad u_{i} := e; \ y := \tilde{e} \{p_{i}\} \quad (17)$$

$$\models \{q_{0} \land e \neq \text{null} \land \forall 1 \leq j \leq n. \neg \text{hastype}(e, c_{j})\} \quad \text{exc} := \text{head}(\text{exc}) \circ e \{q_{1}\} \quad (18)$$

$$\models \{q_{0} \land e \neq \text{null} \land \forall 1 \leq j \leq n. \neg \text{hastype}(e, c_{j})\} \quad \text{exc} := \text{head}(\text{exc}) \circ e; \ y := \tilde{e} \{p_{\text{fin}}\} . \quad (19)$$

For statements \(\{q_{0}\} \text{throw } e \{q_{1}\}^{\text{throw}} \langle y := \tilde{c} \rangle^{\text{throw}}\) in catch blocks, (18) and (19) are required to hold without the above antecedent and with \(p_{\text{fin}}\) replaced by \(p'_{\text{fin}}\). The above conditions (16)-(19) should hold also for statements of the form \(\{q_{0}\}^{\text{exc}} \{q_{1}\}^{\text{rethrow}} \langle y := \tilde{c} \rangle^{\text{rethrow}}\), where the expression \(e\) in the conditions is replaced by \(\top\). Finally, for statements of the form \(\{q_{0}\} \text{throw } e \{q_{1}\}^{\text{throw}} \langle y := \tilde{c} \rangle^{\text{throw}}\) outside try-catch-finally blocks in a run-method with body \(stm'\), return, (18) and (19) should hold without the above antecedent, with \(p_{\text{fin}}\) replaced by \(p_{\text{pre}}(\text{return})\), and without the update of \(\text{exc}\). The above conditions must hold also for all statements \(\{q_{0}\} \{q_{1}\}^{\text{rethrow}} \langle y := \tilde{c} \rangle^{\text{rethrow}}\), where the expression \(e\) in the conditions is replaced by \(\top\).

**The cooperation test** To cover exception handling, we extend the cooperation test conditions for \(Java_{\text{synch}}\) with additional conditions, collected in the cooperation test for exception handling. The cooperation test for exception handling covers exception throwing if it is not in the scope of any try-catch-finally block, i.e., if it causes the control to return to the caller configuration.

Assume a method call and a throw statement outside any try-catch-finally block in the invoked method:

**caller:** \(u_{\text{rel}} := e_{0}.m(\tilde{c}) \ldots \{p_{1}\}^{\text{wait}} \quad \{p_{2}\}^{\text{ret}} \langle y_{4} := \tilde{e}_{4} \rangle^{\text{ret}} \{p_{3}\}^{\text{exc}} \ldots\)

**callee:** \(\ldots \{q_{1}\} \quad \text{throw } e \{q_{2}\}^{\text{throw}} \langle y_{3} := \tilde{e}_{3} \rangle^{\text{throw}} \ldots\)

We assume that the global invariant, the precondition \(q_{1}\) of the throw statement, and the assertion \(p_{1}\) of the caller at the control point waiting for return hold prior to exception throwing. Exception throwing communicates the identity of the thrown exception. Directly after exception throwing, the preconditions \(p_{2}\) and \(q_{2}\) of the corresponding observations must hold, as required by Condition (20) of the cooperation test below. After the throw statement, its observation, and the observation of the caller have been executed, the global invariant and the postcondition \(p_{3}\) of the caller observation is required to hold, as
5.3 The proof system

formalized in Condition (21). Note that the control point after the callee observation is not reachable, thus the assertion at this point is not required to hold.

Let the fresh logical variables \( z \) and \( z' \) denote the caller respectively the callee object. Since these objects are in general different, the cooperation test is formulated in the global language. Local assertions are expressed in the global language using the lifting substitution. For example, the assertion \( p_1 \) of the caller is expressed on the global level by \( P_1(z) = p_1[z/\text{this}] \). To distinguish local variables of caller and callee, we rename those of the callee; the result we denote by primed variables, expressions, and assertions. For example, to reason about \( q_1 \) in the cooperation test we rename all local variables in \( q_1 \) resulting in \( q'_1 \), where \( Q'_1(z') = q'_1[z'/\text{this}] \) is \( q'_1 \) expressed in the global language.

That the identity of the thrown exception is stored in the local variable top of the caller is represented by the assignment \( \text{top} := E'(z') \). The callee and the caller observations are represented by the assignments \( \bar{y}_3 := E_3'(z') \) and \( z \bar{y}_4 := E_4(z) \), respectively. Note that if the invoked method is synchronized, then the observation \( z', \bar{y}_3 := E_3'(z') \) decrements the value of the lock of \( z' \) by the built-in augmentation.

We use the assertion \( \text{comm} \) to express that the local configurations described by \( p_1 \) and \( q_1 \) are indeed communication partners: By \( E_0(z) = z' \) we require that the value of \( z' \) is indeed the callee object of the invocation \( e_0.m(\bar{e}) \). Remember that method call statements must not contain instance variables, and that formal parameters must not be assigned to. That means, the values of \( e_0 \), and the values of the formal and actual parameters do not change during method evaluation. The assertion \( \bar{u}' = E(z) \) states that the values of the formal and of the actual parameters agree, which implies that the primed built-in auxiliary formal parameter \( \bar{c}' \) of the callee stores \((z,\text{conf,thread})\) identifying the caller. I.e., the assertion \( E_0(z) = z' \land \bar{u}' = E(z) \) assures that the local configurations are in caller-callee relationship. Furthermore, \( E'(z') \neq \text{null} \) expresses that the exception to be thrown is not the null reference, i.e., that exception throwing is enabled.

Definition 5.2. (Cooperation test: Exception handling)

A proof outline satisfies the cooperation test for exception handling, if for all statements \( u_{\text{rel}} := e_0.m(\bar{e}) \langle \text{stm} \rangle \text{call} \{ p_1 \} \langle \text{wait} \rangle \{ p_2 \} \langle \text{ret} \rangle \{ \bar{y}_4 := \bar{e}_4 \} \langle \text{ret} \rangle \{ p_3 \} \langle \text{exc} \rangle \rangle \) (or such without receiving a value) occurring in class \( c \) with \( m \neq \text{start} \) and \( e_0 \) of type \( c' \), and for all \( \{ q_1 \} \langle \text{throw} \rangle e \{ q_2 \} \langle \text{throw} \rangle \{ \bar{y}_3 := \bar{e}_3 \} \langle \text{throw} \rangle \) in \( m(\bar{u}) \) of \( c' \) which are not inside any try-catch-finally statement,

\[
|_{\text{g}} \quad \{ \text{GI} \land P_1(z) \land Q'_1(z') \land \text{comm} \} \quad \text{top} := E'(z') \quad \{ P_2(z) \land Q_2(z') \} \quad \text{and (20)}
\]

\[
|_{\text{g}} \quad \{ \text{GI} \land P_1(z) \land Q'_1(z') \land \text{comm} \} \quad \text{top} := E'(z'); \quad z'.\bar{y}_3 := E_3'(z'); \quad z.\bar{y}_4 := E_4(z) \quad \text{(21)}
\]

must hold with distinct fresh logical variables \( z \in \text{LVar}^c \) and \( z' \in \text{LVar}^{c'} \), and with \( \text{comm} \) given by \( E_0(z) = z' \land \bar{u}' = E(z) \land E'(z') \neq \text{null} \land \bar{u} \neq \text{null} \land z' \neq \text{null} \).

Furthermore, the same conditions must hold also for statements of the form \( \{ q_1 \} \langle \text{exc} \rangle \{ q_2 \} \langle \text{rethrow} \rangle \{ \bar{y}_3 := \bar{e}_3 \} \langle \text{rethrow} \rangle \) under the same requirements, where \( e \) in the conditions is replaced by top.

6. Soundness and completeness

This section explains the corner points of soundness and completeness of the proof method. For the formal proofs see [1, 3].
Given a program together with its annotation, the proof system stipulates a number of induction 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. For convenience, let us introduce the following notations: Given a program $\text{prog}$, we will write $\varphi_{\text{prog}}$ or just $\varphi$ for its annotation, and write $\text{prog} \models \varphi$, if $\text{prog}$ satisfies all requirements stated in the assertions, and $\text{prog}' \vdash \varphi'$, if $\text{prog}'$ with annotation $\varphi'$ satisfies the verification conditions of the proof system:

**Definition 6.1.** Given a program $\text{prog}$ with annotation $\varphi$, then $\text{prog} \models \varphi$ iff for all reachable configurations $\langle T, \sigma \rangle$ of $\text{prog}$, for all $(\alpha, \tau, \text{stm}) \in T$, and for all logical environments $\omega$ referring only to values existing in $\sigma$:

1. $\omega, \sigma(\alpha), \tau \models_{L} \text{pre}(\text{stm})$, and
2. $\omega, \sigma \models_{G} \text{GI}$.

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

3. $\omega, \sigma(\beta), \tau' \models_{L} \text{I}_{c}$.

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

In the following sections we discuss the basic ideas of the soundness and completeness proofs.

### 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. The following theorem states the soundness of the proof method.

**Theorem 6.1. (Soundness)**

Let $\text{prog}'$ be a proof outline with annotation $\varphi_{\text{prog}'}$.

If $\text{prog}' \vdash \varphi_{\text{prog}'}$, then $\text{prog} \models \varphi_{\text{prog}'}$.

The soundness proof is basically an induction on the length of computation, simultaneously on all three parts from Definition 6.1. 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 kind of the next computation step: If the computation step executes an assignment, then 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. For communication, invariance for the executing partners and the global invariant is shown using the cooperation test for communication. Exception handling and 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.
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:

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

Given a program satisfying an annotation $\text{prog} \models \varphi_{\text{prog}}$, the consequent can be uniformly shown, i.e., independently of the given assertional part $\varphi_{\text{prog}}$, by instantiating $\varphi_{\text{prog}'}$ to the strongest annotation still provable, thereby discharging the last clause $\models \varphi_{\text{prog}'} \rightarrow \varphi_{\text{prog}}$. 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 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 6.3 below), and finally

3. to show that this augmentation indeed satisfies the verification conditions.

We begin with the augmentation, using the transformation from Section 5.3 as starting point, where the programs are augmented with the specific auxiliary variables. To facilitate reasoning, we introduce an additional auxiliary local variable $\text{loc}$, 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 $\text{loc} := l$, 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 $l \equiv \text{stm}$ if $l$ represents the control point in front of $\text{stm}$.

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

The assertional 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 and the external behavior in the communication history.

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 keeps information about the kind of communication, the communicated values, and the identity of the communication partners involved. For the kind of 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 $\bigcup_{c \in C} \{\text{new}^c\} \cup \bigcup_{m \in M} \{!m, ?m\} \cup \{!\text{return}, ?\text{return}, !\text{throw}, ?\text{throw}\}$ of constants for this purpose, where $C$ and $M$ 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 don’t 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 [5] for such a compositional semantics.

**Definition 6.2. (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
   \[
   h_{inst} := h_{inst} \circ ((\vec{x}, \vec{v})[\vec{e}/\vec{y}]) ,
   \]
   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 get extended with the assignment
   \[
   h_{inst} := \text{if } (e_0 = \text{this}) \text{ then } h_{inst} \text{ else } h_{inst} \circ ((\vec{x}, \vec{v})[\vec{e}/\vec{y}]) \text{ fi} ,
   \]
   instead, if \( m \neq \text{start} \). For \( e_0.start(\vec{e}'); \langle \vec{y} := \vec{e}' \rangle \text{call} \) we use the same update with the condition \( e_0 = \text{this} \) replaced by \( e_0 = \text{this} \wedge \neg \text{started} \).

2. Every observation of communication, object creation, or of a throw statement outside try-catch-finally blocks in a method different from \( \text{run} \) gets extended by
   \[
   h_{comm} := \text{if } (\text{partner} = \text{this}) \text{ then } h_{comm} \circ (\text{sender}, \text{receiver}, \text{values}) \text{ fi} ,
   \]
   where the expressions \( \text{partner}, \text{sender}, \text{receiver}, \text{values} \) depend on the kind of communication as follows:

<table>
<thead>
<tr>
<th>communication</th>
<th>partner</th>
<th>sender</th>
<th>receiver</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>new ( c )</td>
<td>null</td>
<td>this</td>
<td>null</td>
<td>( \text{new } c ), \text{thread}</td>
</tr>
<tr>
<td>( u_{ret} := e_0.m(\vec{e}) ) receive return</td>
<td>( e_0 )</td>
<td>\text{this}</td>
<td>( e_0 )</td>
<td>\text{this}</td>
</tr>
<tr>
<td>( e_0 )</td>
<td>\text{this}</td>
<td>( e_0 )</td>
<td>\text{this}</td>
<td>\text{if } \text{top} = \text{null} \text{ then } \text{return } u_{ret}, \text{thread} \text{ else } ? \text{return } \text{top}, \text{thread} \text{ fi}</td>
</tr>
<tr>
<td>call ( m(\vec{u}) ) receive ( e_{ret} )</td>
<td>\text{caller}_{\text{obj}}</td>
<td>\text{caller}_{\text{obj}}</td>
<td>\text{this}</td>
<td>\text{caller}_{\text{obj}}</td>
</tr>
<tr>
<td>throw ( e )</td>
<td>\text{caller}_{\text{obj}}</td>
<td>\text{this}</td>
<td>\text{caller}_{\text{obj}}</td>
<td>\text{! throw } e, \text{thread}</td>
</tr>
</tbody>
</table>

with \( \text{caller}_{\text{obj}} \) given by the first component of the variable \( \text{caller} \).

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 by the local variables \( \text{thread} \) and \( \text{conf} \), 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 \((\sigma_{inst}, \tau)\) 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.
6.2 Completeness

Definition 6.3. (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 \( \text{Val}(\sigma) = \text{Val}(\sigma') \), and for all \( \alpha \in \text{Val}(\sigma), \sigma(\alpha)(h_{\text{comm}}) = \sigma'(\alpha)(h_{\text{comm}}) \).

2. For each class \( c \), let \( \omega, \sigma_{\text{inst}}, \tau \models_{\mathcal{L}} I_c \) iff there is a reachable \( \langle T, \sigma \rangle \) such that \( \sigma(\alpha) = \sigma_{\text{inst}} \), where \( \alpha = \sigma_{\text{inst}}(\text{this}) \). 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, \( \omega, \sigma_{\text{inst}}, \tau \models_{\mathcal{L}} \text{pre}(stm) \) iff there is a reachable \( \langle T, \sigma \rangle \) with \( \sigma(\alpha) = \sigma_{\text{inst}} \) for \( \alpha = \sigma_{\text{inst}}(\text{this}) \), and \( \langle \alpha, \tau, stm; stm' \rangle \in T \).

4. For preconditions \( p \) of observations observing a statement \( stm \) which is not an assignment, let \( \omega, \sigma_{\text{inst}}, \tau \models_{\mathcal{L}} p \) iff there is a reachable \( \langle T, \sigma \rangle \) with \( \sigma(\alpha) = \sigma_{\text{inst}} \) for \( \alpha = \sigma_{\text{inst}}(\text{this}) \), and with \( \langle \alpha, \tau', \text{stm}; \text{stm'} \rangle \in T \) enabled to execute resulting in the local state \( \tau \) directly after the execution of the statement but before its observation.

For the reception of a method call, instead of the existence of the enabled \( \langle \alpha, \tau', \text{stm}; \text{stm'} \rangle \in T \), we require that a call of the method of \( \alpha \) is enabled in \( \langle T, \sigma \rangle \) with resulting callee local state \( \tau \) directly after communication\(^{15}\).

It can be shown that these assertions are expressible in the assertion language [56]. The augmented program together with the above annotation build a proof outline that we denote by \( \text{prog}' \).

What remains to be shown for completeness is that the proof outline \( \text{prog}' \) 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 \( \text{prog}' \) 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 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 to uniquely determine the way a configuration has been reached; in the case of instance history, uniqueness of course, only as far as the chosen instance is concerned. This property is stated formally in the following local merging lemma.

Lemma 6.1. (Local merging lemma)

Assume two reachable global configurations \( \langle T_1, \sigma_1 \rangle \) and \( \langle T_2, \sigma_2 \rangle \) of \( \text{prog}' \) and \( \langle \alpha, \tau, \text{stm} \rangle \in T_1 \) with \( \alpha \in \text{Val}(\sigma_1) \cap \text{Val}(\sigma_2) \). Then \( \sigma_1(\alpha)(h_{\text{inst}}) = \sigma_2(\alpha)(h_{\text{inst}}) \) implies \( \langle \alpha, \tau, \text{stm} \rangle \in T_2 \).

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 agreeing on the communication histories of the objects. This information must be enough to ensure common reachability. Such a common

\(^{15}\)For the precondition of the observation \( \text{stm} \) 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 \( \text{stm} \).
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. (Global merging lemma)**

Assume two reachable global configurations \( \langle T_1, \sigma_1 \rangle \) and \( \langle T_2, \sigma_2 \rangle \) of \( \text{prog'} \) and \( \alpha \in \text{Val}(\sigma_1) \cap \text{Val}(\sigma_2) \) with the property \( \sigma_1(\alpha)(h_{\text{comm}}) = \sigma_2(\alpha)(h_{\text{comm}}) \). Then there exists a reachable configuration \( \langle T, \sigma \rangle \) with \( \text{Val}(\sigma) = \text{Val}(\sigma_2), \sigma(\alpha) = \sigma_1(\alpha) \), and \( \sigma(\beta) = \sigma_2(\beta) \) for all \( \beta \in \text{Val}(\sigma_2) \setminus \{\alpha\} \).

Note that together with the local merging lemma this implies that all local configurations in \( \langle T_1, \sigma_1 \rangle \) executing in \( \alpha \) 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. (Completeness)**

For a program \( \text{prog} \), the proof outline \( \text{prog'} \) satisfies the verification conditions of the proof system from Section 5.3.

7. Conclusion

In this work we presented a tool-supported assertional proof method for a Java sublanguage including multithreading and exception handling. We introduced the language and the proof system incrementally in four 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 and exception handling. The resulting proof system is sound and complete.

Tool support is given by the proof condition generator Verger. The tool takes an augmented and annotated Java program, a so-called proof outline, as input and generates the verification conditions, which assure invariance of the annotation. We use the theorem prover PVS to verify the conditions.

Future work 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 and subtyping. We are also interested on the development of a compositional proof system.

References


6.2 Completeness


