

A Deductive Proof System for Multithreaded Java with Exceptions*

Erika Ábrahám[†]

Albert-Ludwigs-University Freiburg, Germany
eab@informatik.uni-freiburg.de
e.abraham@fz-juelich.de

Willem-Paul de Roever

Christian-Albrechts-University, Kiel, Germany
wpr@informatik.uni-kiel.de

Frank S. de Boer

CWI, Amsterdam, The Netherlands
F.S.de.Boer@cwi.nl

Martin Steffen

University of Oslo, Norway
msteffen@ifi.uio.no

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

*Part of this work has been financially supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services*. and the NWO/DFG project *Mobi-J* (RO 1122/9- $\{1,2,4\}$).

[†]Address for correspondence: Albert-Ludwigs-University Freiburg, Fahnbergplatz 79085 Freiburg, Germany

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 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* [43], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [22, 23]. 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* [43, 32], 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 [12] and in [32] 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 losing 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 [44] to verify the conditions, for which we only need to encode the semantics of the assertion language. The proof system here deals with *partial correctness* properties only, i.e., we do not address termination properties.

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.

In this paper we concentrate on the theoretical background. For application examples demonstrating how to use the proof system, see [1, 3].

1.1. Related work

This work extends earlier results. In [6] we develop a proof system for a concurrent sublanguage of *Java*, but without reentrant monitors. Reentrant synchronization was incorporated in [8]; the work [2] 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 [7]. In [7] we also introduce proof conditions for deadlock freedom. The work is summarized in Ábrahám's PhD thesis [1] and the theoretical aspects in [5]. We formalize the semantics of our programming language in a compositional manner in [4]. 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. [9, 54, 21]). The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sub-languages. See [31] for a recent and extensive survey over different approaches towards the specification and verification of object-oriented programs (and concentrating on sequential 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 [17] 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 [47, 19, 18] and especially the PhD thesis of Pierik [46] 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) [33] 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 [15] translates programs and their specifications into *PVS* [28] and *Isabelle/HOL* [14]. 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 [24]. LOOP specifications formalized in *JML* are represented in HOL by a set of proof rules [30]. Jacobs presents also a coalgebraic view of exceptions in [27]. Modeling inheritance in higher order logic is the topic of [25]. The LOOP tool and methodology has been applied to several case studies; see e.g. [53, 52, 16, 26, 29].

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 [13] is concerned with the formalization of various aspects of *Java* in the theorem prover *Isabelle/HOL* [45]. Nipkow and von Oheimb [36, 41] prove type soundness of their *Java_{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_{light}*, von Oheimb presents an axiomatic semantics [38, 39], and proves soundness and completeness of the latter with respect to the operational semantics. With μ *Java*, Nipkow et al. [37] 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 [37], von Oheimb [40] presents a Hoare-style calculus for a *JavaCard* subset and proves soundness and completeness in *Isabelle/HOL*. Nipkow [35] selects some of the technically difficult language features and deals with their Hoare logic in isolation. The combination of [40] and [35] in one language (NanoJava) is formulated in [42].

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 [48, 49, 50, 51]. 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. The verification conditions in the above sections are formulated as standard Hoare-triples. Section 6 defines the formal semantics of Hoare-triples, given by means of a weakest precondition calculus, and reformulates the verification conditions. Soundness and completeness are discussed in Section 7. Section 8 contains some concluding remarks.

2. The sequential language

In this section we introduce a sequential sublanguage $Java_{seq}$ of *Java*. We define its syntax in Section 2.1, and its semantics in Section 2.2. After defining the assertion 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 $Java_{seq}$ 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 $Java_{seq}$.

2.1. Syntax

$Java_{seq}$ is a strongly typed language; besides class types c , it supports booleans `Bool` and integers `Int` as primitive types, and pairs $t \times t$ and lists `list` t as composite types. The type of methods without return

$$\begin{aligned}
exp & ::= x \mid u \mid \text{this} \mid \text{null} \mid f(exp, \dots, exp) \\
exp_{ret} & ::= \epsilon \mid exp \\
stm & ::= x := exp \mid u := exp \mid u := \text{new}^c \\
& \quad \mid u := exp.m(exp, \dots, exp) \mid exp.m(exp, \dots, exp) \\
& \quad \mid \epsilon \mid stm; stm \mid \text{if } exp \text{ then } stm \text{ else } stm \text{ fi} \mid \text{while } exp \text{ do } stm \text{ od} \dots \\
meth & ::= m(u, \dots, u) \{ stm; \text{return } exp_{ret} \} \\
meth_{run} & ::= \text{run}() \{ stm; \text{return} \} \\
class & ::= \text{class } c \{ meth \dots meth \} \\
class_{main} & ::= c \{ meth \dots meth meth_{run} \} \\
prog & ::= \langle class \dots class class_{main} \rangle
\end{aligned}$$
Table 1. $Java_{seq}$ abstract syntax

value is Void. Since $Java_{seq}$ is strongly typed, all program constructs of the abstract syntax are silently assumed to be well-typed. In other words, we work with a type-annotated abstract syntax where we omit the explicit mentioning of types when this causes no confusion.

For each type, the corresponding value domain is equipped with a standard set of operators with typical element f . Each operator f has a unique type $t_1 \times \dots \times t_n \rightarrow t$ 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* $x \in IVar$ and *local (temporary) variables* $u \in TVar$. 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 $Var = IVar \dot{\cup} TVar$ for the set of program variables with typical element y , where $\dot{\cup}$ is the disjoint union operator.

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

Besides using instance and local variables, *expressions* $exp \in Exp$ are built from the self-reference `this`, the empty reference `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, $Java_{seq}$ 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* $stm \in Stm$, we allow assignments, object creation, method invocation, and standard control constructs like sequential composition, conditional statements, and iteration. We write ϵ for the empty statement.

A *method definition* $m(u_1, \dots, u_n) \{ stm; \text{return } e_{ret} \}$ specifies the method's name m , a list of formal parameters u_1, \dots, u_n , and a method body of the form $stm; \text{return } e_{ret}$, 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 $Meth_c$ contains the methods of class c . We denote the body of method m of class c by $body_{m,c}$.

Sometimes we explicitly mention the types of formal parameters and of the return value in *Java*-style $t\ m(t_1\ u_1, \dots, t_n\ u_n)\{body_{m,c}\}$.

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 $class_{main}$ defines by its run-method the entry point of the program execution. We call the body of the run-method of the main class the *main statement* of the program.¹ The run-method cannot be called.

The set $IVar_c$ of instance variables of a class c is given implicitly by the instance variables occurring in the class; the set of local variables of method declarations is given similarly. In the examples we 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, \dots, e_n in a method invocation $e_0.m(e_1, \dots, e_n)$ 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 losing 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 α, β, \dots denote infinite sets of *object identifiers*. The value of null of type c is $null^c \notin Val^c$. [This is the semantic analogue to monomorphism.] In general we will just write *null*, when c is clear from the context. We define Val_{null}^c as $Val^c \dot{\cup} \{null^c\}$, 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 $\bigcup_t Val_{null}^t$. 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 $Init$.²

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

²In *Java*, this is a "final" instance variable, which for instance implies, it cannot be assigned to.

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 $\tau_{m,c}^{init}$ assigns to each local variable u from its domain the value $Init(u)$. A *local configuration* (α, τ, stm) of a method of an object $\alpha \neq null$ specifies, in addition to its local state τ , its point of execution represented by the statement stm . A *thread configuration* $\xi = (\alpha_0, \tau_0, stm_0)(\alpha_1, \tau_1, stm_1) \dots (\alpha_n, \tau_n, stm_n)$ is a stack of local configurations, representing the 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* $\sigma_{inst} \in \Sigma_{inst}$ of type $IVar \dot{\cup} \{\text{this}\} \rightarrow Val_{null}$; we require that this is in the domain $dom(\sigma_{inst})$ of σ_{inst} . We write σ_{inst}^c to denote states of instances of class c . The semantics will maintain $\sigma_{inst}^c(\text{this}) \in Val^c$ as invariant. The initial instance state $\sigma_{inst}^{c,init}$ assigns a value from Val^c to this, and to each of its remaining instance variables x the value $Init(x)$.

A *global state* $\sigma \in \Sigma$ of type $(\bigcup_c Val^c) \rightarrow \Sigma_{inst}$ stores for each currently *existing* object, i.e., an object belonging to the domain of σ , its instance state. The set of existing objects of type c in a state σ is given by $Val^c(\sigma)$, and $Val_{null}^c(\sigma) = Val^c(\sigma) \dot{\cup} \{null^c\}$. For the remaining types, $Val^t(\sigma)$ and $Val_{null}^t(\sigma)$ are defined correspondingly. We refer to the set $\bigcup_t Val^t(\sigma)$ by $Val(\sigma)$; $Val_{null}(\sigma)$ denotes $\bigcup_t Val_{null}^t(\sigma)$. The instance state of an object $\alpha \in Val(\sigma)$ 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., $\sigma(\alpha)(x) \in Val_{null}(\sigma)$ for all classes c , objects $\alpha \in Val^c(\sigma)$, and instance variables $x \in IVar_c$. This will be an invariant of the operational semantics of the next section.

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

The semantic function $\llbracket _ \rrbracket_{\mathcal{E}}^- : (\Sigma_{inst} \times \Sigma_{loc}) \rightarrow (Exp \rightarrow Val_{null})$ evaluates in the context of an *instance local state* (σ_{inst}, τ) expressions containing variables from $dom(\sigma_{inst}) \cup dom(\tau)$: Instance variables x and local variables u are evaluated to $\sigma_{inst}(x)$ and $\tau(u)$, respectively; this evaluates to $\sigma_{inst}(\text{this})$, 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 τ on the values of all other variables; $\sigma_{inst}[x \mapsto v]$ is defined analogously, where $\sigma[\alpha.x \mapsto v]$ results from σ by assigning v to the instance variable x of object α . We use these operators analogously for vectors of variables. We use $\tau[\vec{y} \mapsto \vec{v}]$ also for arbitrary variable sequences, where instance variables are untouched; $\sigma_{inst}[\vec{y} \mapsto \vec{v}]$ and $\sigma[\alpha.\vec{y} \mapsto \vec{v}]$ are analogous. Finally for global states, $\sigma[\alpha \mapsto \sigma_{inst}]$ equals σ except on α ; note that in case $\alpha \notin Val(\sigma)$, the operation extends the set of existing objects by α , which has its instance state initialized to σ_{inst} .

$$\begin{aligned}
\llbracket x \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau} &= \sigma_{inst}(x) \\
\llbracket u \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau} &= \tau(u) \\
\llbracket \text{this} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau} &= \sigma_{inst}(\text{this}) \\
\llbracket \text{null} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau} &= \text{null} \\
\llbracket f(e_1, \dots, e_n) \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau} &= f(\llbracket e_1 \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}, \dots, \llbracket e_n \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau})
\end{aligned}$$

Table 2. Semantics of program expressions

2.2.2. Operational semantics

The operational semantics of $Java_{seq}$ 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 \longrightarrow , let us start by defining the starting point of a program. The initial configuration $\langle T_0, \sigma_0 \rangle$ of a program satisfies $dom(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma_{inst}^{c, init}[\text{this} \mapsto \alpha]$, and $T_0 = \{(\alpha, \tau_{run, c}^{init}, body_{run, c})\}$, where c is the main class, and $\alpha \in Val^c$.

We call a configuration $\langle T, \sigma \rangle$ of a program *reachable* iff there is a computation $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle T, \sigma \rangle$ such that $\langle T_0, \sigma_0 \rangle$ is the initial configuration of the program and \longrightarrow^* the reflexive transitive closure of \longrightarrow . A local configuration $(\alpha, \tau, stm) \in T$ is *enabled* in $\langle T, \sigma \rangle$, if it can be executed, i.e., if there is a computation step $\langle T, \sigma \rangle \rightarrow \langle T', \sigma' \rangle$ executing stm in the local state τ and object α .

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

2.3. The assertion language

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

$\frac{}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, x := e; stm) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, stm) \}, \sigma[\alpha.x \mapsto \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau}] \rangle}$	ASS_{inst}
$\frac{}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, u := e; stm) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau[u \mapsto \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau}], stm) \}, \sigma \rangle}$	ASS_{loc}
$\frac{\beta \in Val^c \setminus Val(\sigma) \quad \sigma_{inst} = \sigma_{inst}^{c, init}[\text{this} \mapsto \beta] \quad \sigma' = \sigma[\beta \mapsto \sigma_{inst}]}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, u := \text{new}^c; stm) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau[u \mapsto \beta], stm) \}, \sigma' \rangle}$	NEW
$\frac{m(\vec{u})\{ body \} \in Meth_c, \beta = \llbracket e_0 \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \in Val^c(\sigma) \quad \tau' = \tau_{m,c}^{init}[\vec{u} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau}]}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, u := e_0.m(\vec{e}); stm) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{receive } u; stm) \circ (\beta, \tau', body) \}, \sigma \rangle}$	CALL
$\frac{\tau'' = \tau[u_{ret} \mapsto \llbracket e_{ret} \rrbracket_{\mathcal{E}}^{\sigma(\beta), \tau'}]}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{receive } u_{ret}; stm) \circ (\beta, \tau', \text{return } e_{ret}) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau'', stm) \}, \sigma \rangle}$	RETURN
$\frac{}{\langle T \dot{\cup} \{ (\alpha, \tau, \text{return}) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ (\alpha, \tau, \epsilon) \}, \sigma \rangle}$	RETURN_{run}

Table 3. $Java_{seq}$ operational semantics

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 *Vergier* 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 $e, e' \dots$ 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, \dots , 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 quantifica-

$exp_l ::= z \mid x \mid u \mid \text{this} \mid \text{null} \mid f(exp_l, \dots, exp_l)$	$e \in LExp$
$ass_l ::= exp_l \mid \neg ass_l \mid ass_l \wedge ass_l$	
$\quad \mid \exists z. ass_l \mid \exists z \in exp_l. ass_l \mid \exists z \sqsubseteq exp_l. ass_l$	$p \in LAss$
$exp_g ::= z \mid \text{null} \mid f(exp_g, \dots, exp_g) \mid exp_g.x$	$E \in GExp$
$ass_g ::= exp_g \mid \neg ass_g \mid ass_g \wedge ass_g \mid \exists z. ass_g$	$P \in GAss$

Table 4. Syntax of assertions

tion is not allowed, as for those types the existence of a value dynamically depends on the *global* state, something one cannot speak about on the local level, or more formally: Disallowing unrestricted quantification for object types ensures that the value of a local assertion indeed only depends on the values of the instance and local variables, but not on the global state. Nevertheless, one can assert the existence of objects on the local level satisfying a predicate, provided one is explicit about the set of objects to range over. Thus, the restricted quantifications $\exists z \in e. p$ and $\exists z \sqsubseteq e. p$ assert the existence of an element, respectively, the existence of a subsequence of a given sequence e , for which a property p holds.

Global expressions $exp_g \in GExp$, with typical elements E, E', \dots , 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, \dots , 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 $Val_{null}(\sigma)$ 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 $LVar \rightarrow Val_{null}$, assigning values to logical variables. We denote by $\omega[\vec{z} \mapsto \vec{v}]$ the logical environment that assigns the values \vec{v} to the variables \vec{z} , and agrees with ω on all other variables. Similarly to local and instance state updates, the occurrence of instance and local variables in \vec{z} is without effect. For a logical environment ω and a global state σ we say that ω refers only to values existing in σ , if $\omega(z) \in Val_{null}(\sigma)$ for all $z \in dom(\omega)$. This property matches with the definition of quantification which ranges only over existing values and *null*, and with the fact that in reachable configurations local variables may refer only to existing values or to *null*.

The semantic function $\llbracket - \rrbracket_{\mathcal{L}}^{\omega, \tau}$ of type $(\Omega \times \Sigma_{inst} \times \Sigma_{loc}) \rightarrow (LExp \cup LAss \rightarrow Val_{null})$ evaluates local expressions and assertions in the context of a logical environment ω and an instance local state (σ_{inst}, τ) (see Table 5). The evaluation function is defined for expressions and assertions that contain only variables from $dom(\omega) \cup dom(\sigma_{inst}) \cup dom(\tau)$. The instance local state provides the context for giving meaning to programming language expressions as defined by the semantic function $\llbracket - \rrbracket_{\mathcal{E}}^{\omega, \tau}$; the

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

Table 5. Local evaluation

logical environment evaluates logical variables. An unrestricted quantification $\exists z. p$ with $z \in LVar^t$ is evaluated to true in the logical environment ω and instance local state (σ_{inst}, τ) if and only if there exists a value $v \in Val^t$ such that p holds in the logical environment $\omega[z \mapsto v]$ and instance local state (σ_{inst}, τ) , where for the type t of z only `Int`, `Bool`, or compound types built from them are allowed. The evaluation of a restricted quantification $\exists z \in e. p$ with $z \in LVar^t$ and $e \in LExp^{\text{list } t}$ is defined analogously, where the existence of an element in the sequence is required. An assertion $\exists z \sqsubseteq e. p$ with $z \in LVar^{\text{list } t}$ and $e \in LExp^{\text{list } t}$ states the existence of a subsequence of e for which p holds. In the following we also write $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ for $\llbracket p \rrbracket_{\mathcal{L}}^{\omega, \sigma_{inst}, \tau} = \text{true}$. By $\models_{\mathcal{L}} p$ we express that $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ holds for arbitrary logical environments, instance states, and local states.

Since *global* assertions do not contain local variables and non-qualified references to instance variables, the global assertion semantics does not refer to instance local states but to global states. The semantic function $\llbracket _ \rrbracket_{\mathcal{G}}$ of type $(\Omega \times \Sigma) \rightarrow (GExp \cup GAss \rightarrow Val_{\text{null}})$, shown in Table 6, gives meaning to global expressions and assertions in the context of a global state σ and a logical environment ω . To be well-defined, ω is required to refer only to values existing in σ , and the expression respectively assertion may only contain free variables³ from the domain of ω . Logical variables, `null`, and operator expressions are evaluated analogously to local assertions. The value of a global expression $E.x$ is given by the value of the instance variable x of the object referred to by the expression E . The evaluation of an expression $E.x$ is defined only if E refers to an object existing in σ . Note that when E and E' refer to the same object, that is, E and E' are *aliases*, then $E.x$ and $E'.x$ denote the same variable. The semantics of negation and conjunction is standard. A quantification $\exists z. P$ with $z \in LVar^t$ evaluates to true in the context of ω and σ if and only if P evaluates to true in the context of $\omega[z \mapsto v]$ and σ , for some value $v \in Val_{\text{null}}^t(\sigma)$. Note that quantification over objects ranges over the set of *existing* objects and *null*, only.

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

³In global expressions $E.x$ we treat x as a bound variable.

$$\begin{array}{l}
\llbracket z \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \omega(z) \\
\llbracket \text{null} \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{null} \\
\llbracket f(E_1, \dots, E_n) \rrbracket_{\mathcal{G}}^{\omega, \sigma} = f(\llbracket E_1 \rrbracket_{\mathcal{G}}^{\omega, \sigma}, \dots, \llbracket E_n \rrbracket_{\mathcal{G}}^{\omega, \sigma}) \\
\llbracket E.x \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \sigma(\llbracket E \rrbracket_{\mathcal{G}}^{\omega, \sigma})(x) \\
(\llbracket \neg P \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{true}) \text{ iff } (\llbracket P \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{false}) \\
(\llbracket P_1 \wedge P_2 \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{true}) \text{ iff } (\llbracket P_1 \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{true} \text{ and } \llbracket P_2 \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{true}) \\
(\llbracket \exists z. P \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{true}) \text{ iff } (\llbracket P \rrbracket_{\mathcal{G}}^{\omega[z \mapsto v], \sigma} = \text{true} \text{ for some } v \in \text{Val}_{\text{null}}(\sigma))
\end{array}$$

Table 6. Global evaluation

To express a local property p in the global assertion language, we define the substitution $p[z/\text{this}]$ 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 $p[z/\text{this}]$ as logical variables. Formally, these local variables are replaced by fresh logical variables. We write $P(z)$ for $p[z/\text{this}]$, and similarly for expressions. For unrestricted quantifications $(\exists z'. p)[z/\text{this}]$ the substitution applies to the assertion p . Local restricted quantifications are transformed into global unrestricted ones where the relations \in and \sqsubseteq are expressed at the global level as operators. The main cases of the substitution are defined as follows:

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

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

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

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].

2.4.1. 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*.

Note that augmentation is not optional but necessary for relative completeness when reasoning about computations. For example, without auxiliary variables, in general it would not be possible to describe the input-output behaviour of a method by an assertion, since the input-output relation relates pairs of states, while assertions are evaluated in a single state. To express such dependencies, one can introduce new auxiliary variables to store the input (i.e., parameter) values, and state the property by an assertion evaluated in the output state. Other proof methods may handle augmentation differently: JML for example introduces for this special case of input-output relation of methods a special auxiliary variable “\return” for the return value.

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 $u := new^c \langle \vec{y} := \vec{e} \rangle^{new}$ which attaches the observation to the object creation statement. Observations $\vec{y}_1 := \vec{e}_1$ of a method call and observations $\vec{y}_4 := \vec{e}_4$ of the corresponding reception of a return value are denoted by $u := e_0.m(\vec{e}) \langle \vec{y}_1 := \vec{e}_1 \rangle^{call} \langle \vec{y}_4 := \vec{e}_4 \rangle^{ret}$. The augmentation $\langle \vec{y}_2 := \vec{e}_2 \rangle^{call} stm; return e_{ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{ret}$ of method bodies specifies $\vec{y}_2 := \vec{e}_2$ as the observation of the reception of the method call and $\vec{y}_3 := \vec{e}_3$ as the observation attached to the return statement. Assignments can be observed using $\vec{y} := \vec{e} \langle \vec{y}' := \vec{e}' \rangle^{ass}$. A stand-alone observation not attached to any statement is written as $\langle \vec{y} := \vec{e} \rangle$. It can be inserted at any point in the program.

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

The augmentation does not influence the control flow of the program but enforces a particular scheduling policy. 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*.

As stated above, in the case of method calls the caller and the callee both may make observations. Consequently, in case of self-calls, when caller and callee execute in the same object, we would have to show interference freedom under assignment-pairs, which would increase the complexity of the proof system. To overcome this complication, we require that the caller observation in a self-communication may not change the values of instance variables. Formally, in each observation of a method invocation statement $e_0.m(\vec{e})$, assignments to instance variables must have the form $x := \text{if } e_0 = \text{this then } x \text{ else } e \text{ fi}$.

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 configuration in that object. The callee receives the “return address” as auxiliary formal parameter *caller* of type $\text{Object} \times \text{Int}$, storing the identities of the caller object and the calling local configuration. The run-method of the initial object is executed with the parameter *caller* having the value $(\text{null}, 0)$.

Syntactically, each method declaration $m(\vec{u})\{stm; \text{return } e_{ret}\}$ gets extended by the built-in augmentation to $m(\vec{u}, \text{caller})\{\langle \text{conf}, \text{counter} := \text{counter}, \text{counter} + 1 \rangle^{?call} stm; \text{return } e_{ret}\}$. Correspondingly for method calls $u := e_0.m(\vec{e})$, the actual parameter lists get extended to $u := e_0.m(\vec{e}, (\text{this}, \text{conf}))$. 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 $\{p\} stm \{q\}$ and write $pre(stm)$ and $post(stm)$ to refer to the pre- and the post-condition of a statement. For assertions at auxiliary points we use the following notation: The annotation

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

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

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

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 caller after method call and before return; finally, p_3 specifies the state directly after return but before its observation. The annotation of method bodies $stm; \text{return } e_{ret}$ is as follows:

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

The callee postcondition of the method call is p_1 ; the callee pre- and postconditions 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.

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 Init be a syntactical operator with interpretation Init (see page 396). Given 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 $z \neq \text{null} \wedge \bigwedge_{x \in \text{IVar}_c} z.x = \text{Init}(x)$, 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 $\{p_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{p_3\} \text{stm}; \text{return with local variables } \vec{v}$ without the formal parameters, $z \in \text{LVar}^c$, and $z' \in \text{LVar}^{\text{Object}}$. A proof outline is *initially correct*, if

$$\models_{\mathcal{G}} \{ \text{InitState}(z) \wedge \forall z'. z' = \text{null} \vee z = z' \} \quad \vec{v}, \text{ caller} := \text{Init}(\vec{v}), (\text{null}, 0) \quad \{P_2(z)\} \quad (1)$$

$$\models_{\mathcal{G}} \{ \text{InitState}(z) \wedge \forall z'. z' = \text{null} \vee z = z' \} \quad (2)$$

$$\vec{v}, \text{ caller} := \text{Init}(\vec{v}), (\text{null}, 0); \quad z.\vec{y}_2 := \vec{E}_2(z) \\ \{ GI \wedge P_3(z) \wedge I_c(z) \}$$

⁴The 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.

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

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 [11] 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,

$$\models_{\mathcal{L}} \{p_1\} \vec{y} := \vec{e} \{p_2\}. \quad (3)$$

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 caller_obj be the first and caller_conf the second component of caller. We define $\text{waits_for_ret}(q, \vec{y} := \vec{e})$ 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;
- $\text{conf}' \neq \text{conf} \wedge (\text{this} \neq \text{caller_obj} \vee \text{conf}' \neq \text{caller_conf})$, 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 ,

$$\models_{\mathcal{L}} \{p \wedge I_c\} \vec{y} := \vec{e} \{I_c\}. \quad (4)$$

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

$$\models_{\mathcal{L}} \{p \wedge q' \wedge \text{waits_for_ret}(q, \vec{y} := \vec{e})\} \vec{y} := \vec{e} \{q'\}. \quad (5)$$

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 *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, comm additionally states $\vec{u}' = \vec{E}(z)$, where \vec{u} and \vec{e} are the formal and the actual parameters. Returning from the run-method terminates the executing thread, 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 $\vec{u}' := \vec{E}(z)$, where initialization of the remaining local variables \vec{v} is covered by $\vec{v}' := \text{Init}(\vec{v})$. The assignments $z.\vec{y}'_1 := \vec{E}_1(z)$ and $z'.\vec{y}'_2 := \vec{E}'_2(z')$ stand for the caller and callee observations $\vec{y}'_1 := \vec{e}_1$ and $\vec{y}'_2 := \vec{e}_2$, executed in the objects z and z' , respectively. Note that we rename all local variables of the callee to avoid name clashes.

Definition 2.4. (Cooperation test: Communication)

A proof outline satisfies the *cooperation test for communication*, if

$$\begin{aligned} \models_G \quad & \{GI \wedge P_1(z) \wedge Q'_1(z') \wedge \text{comm} \wedge z \neq \text{null} \wedge z' \neq \text{null}\} \\ & \quad \quad \quad f_{\text{comm}} \\ & \quad \quad \quad \{P_2(z) \wedge Q'_2(z')\} \end{aligned} \quad (6)$$

$$\begin{aligned} \models_G \quad & \{GI \wedge P_1(z) \wedge Q'_1(z') \wedge \text{comm} \wedge z \neq \text{null} \wedge z' \neq \text{null}\} \\ & \quad \quad \quad f_{\text{comm}}; \quad f_{\text{obs1}}; \quad f_{\text{obs2}} \\ & \quad \quad \quad \{GI \wedge P_3(z) \wedge Q'_3(z')\} \end{aligned} \quad (7)$$

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

1. **CALL:** For all statements $\{p_1\} u_{\text{ret}} := e_0.m(\vec{e}) \{p_2\}'^{\text{call}} \langle \vec{y}'_1 := \vec{e}_1 \rangle'^{\text{call}} \{p_3\}'^{\text{wait}}$ (or such without receiving a value) in class c with e_0 of type c' , where method m of c' has body $\{q_2\}'^{\text{call}} \langle \vec{y}'_2 := \vec{e}_2 \rangle'^{\text{call}} \{q_3\}'^{\text{stm}}; \text{return } e_{\text{ret}}$, formal parameters \vec{u} , and local variables \vec{v} except the formal parameters. The callee class invariant is $q_1 = I_{c'}$. The assertion comm is given by $E_0(z) = z'$. Furthermore, f_{comm} is $\vec{u}', \vec{v}' := \vec{E}(z)$, $\text{Init}(\vec{v})$, f_{obs1} is $z.\vec{y}'_1 := \vec{E}_1(z)$, and f_{obs2} is $z'.\vec{y}'_2 := \vec{E}'_2(z')$.
2. **RETURN:** For all $u_{\text{ret}} := e_0.m(\vec{e}) \langle \text{stm} \rangle'^{\text{call}} \{p_1\}'^{\text{wait}} \{p_2\}'^{\text{ret}} \langle \vec{y}'_4 := \vec{e}_4 \rangle'^{\text{ret}} \{p_3\}'$ (or such without receiving a value) occurring in c with e_0 of type c' , such that method m of c' has the return statement $\{q_1\}' \text{return } e_{\text{ret}} \{q_2\}'^{\text{ret}} \langle \vec{y}'_3 := \vec{e}_3 \rangle'^{\text{ret}} \{q_3\}'$, and formal parameter list \vec{u} , the above equations must hold with comm given by $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$, and where f_{comm} is $u_{\text{ret}} := E'_{\text{ret}}(z')$, f_{obs1} is $z'.\vec{y}'_3 := \vec{E}'_3(z')$, and f_{obs2} is $z.\vec{y}'_4 := \vec{E}_4(z)$.
3. **RETURN_{run}:** For $\{q_1\}' \text{return } \{q_2\}'^{\text{ret}} \langle \vec{y}'_3 := \vec{e}_3 \rangle'^{\text{ret}} \{q_3\}'$ occurring in the run-method of the main class, $p_1 = p_2 = p_3 = \text{true}$, comm = true, and furthermore f_{comm} and f_{obs2} are the empty statement, and f_{obs1} is $z'.\vec{y}'_3 := \vec{E}'_3(z')$.

Besides method calls and returns, the cooperation test needs to handle object creation, taking care of the preservation of the global invariant, the postcondition of the new statement and its observation, and the new object's class invariant. We can assume that the precondition of the object creation statement and the global invariant hold in the configuration prior to instantiation. The extension of the global state with a freshly created object is formulated in a *strongest postcondition* style, i.e., it is required to hold immediately *after* the instantiation. We use existential quantification to refer to the old value: z' of type

$LVar^{\text{list Object}}$ represents the existing objects prior to the extension. Moreover, that the created object's identity stored in u is fresh and that the new instance is properly initialized is expressed by the global assertion $\text{Fresh}(z', u)$ defined as $\text{InitState}(u) \wedge u \notin z' \wedge \forall v. v \in z' \vee v = u$ (see page 405 for the definition of InitState). To express that an assertion refers to the set of existing objects *prior* to the extension of the global state, we need to *restrict* any existential quantification in the assertion to range over objects from z' , only. So let P be a global assertion and $z' \in LVar^{\text{list Object}}$ a logical variable not occurring in P . Then $P \downarrow z'$ is the global assertion P with all quantifications $\exists z. P'$ replaced by $\exists z. \text{obj}(z) \subseteq z' \wedge P'$, where $\text{obj}(v)$ denotes the set of objects occurring in the value v . The following lemma formulates the basic property of the projection operator:

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

$$\omega, \sigma \models_G P \quad \text{iff} \quad \omega[z' \mapsto v], \sigma' \models_G P \downarrow z'.$$

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 $\{p_1\} u := \text{new}^c \{p_2\}^{new} \langle \vec{y} := \vec{e} \rangle^{new} \{p_3\}$ in c' :

$$\begin{aligned} \models_G \quad & z \neq \text{null} \wedge z \neq u \wedge \exists z'. (\text{Fresh}(z', u) \wedge (GI \wedge \exists u. P_1(z)) \downarrow z') \\ & \rightarrow P_2(z) \wedge I_c(u) \end{aligned} \quad (8)$$

$$\begin{aligned} \models_G \quad & \{z \neq \text{null} \wedge z \neq u \wedge \exists z'. (\text{Fresh}(z', u) \wedge (GI \wedge \exists u. P_1(z)) \downarrow z')\} \\ & z.\vec{y} := \vec{E}(z) \\ & \{GI \wedge P_3(z)\} \end{aligned} \quad (9)$$

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

3. The concurrent language

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

3.1. Syntax

Expressions and statements can be constructed as in $Java_{seq}$. 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

$$\begin{aligned}
meth & ::= m(u, \dots, u) \{ stm; \text{return } exp_{ret} \} \\
meth_{run} & ::= \text{run}() \{ stm; \text{return} \} \\
class & ::= \text{class } c \{ meth \dots meth \ meth_{run} \ meth_{start} \} \\
class_{main} & ::= class \\
prog & ::= \langle class \dots class \ class_{main} \rangle
\end{aligned}$$
Table 7. $Java_{conc}$ abstract syntax

$$\frac{\beta = \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \in Val^c(\sigma) \quad \neg started(T \cup \{\xi \circ (\alpha, \tau, e.start()); stm\}), \beta}{\langle T \dot{\cup} \{\xi \circ (\alpha, \tau, e.start()); stm\}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{\xi \circ (\alpha, \tau, stm), (\beta, \tau_{run,c}^{init}, body_{run,c})\}, \sigma \rangle} \text{CALL}_{start}$$

$$\frac{\beta = \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \in Val(\sigma) \quad started(T \cup \{\xi \circ (\alpha, \tau, e.start()); stm\}), \beta}{\langle T \dot{\cup} \{\xi \circ (\alpha, \tau, e.start()); stm\}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{\xi \circ (\alpha, \tau, stm)\}, \sigma \rangle} \text{CALL}_{start}^{skip}$$

Table 8. $Java_{conc}$ operational semantics

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 $\text{CALL}_{start}^{skip}$).⁵ This is captured by the predicate $started(T, \beta)$ which holds iff there exists a stack $(\alpha_0, \tau_0, stm_0) \dots (\alpha_n, \tau_n, stm_n) \in T$ such that $\beta = \alpha_0$. A thread ends its lifespan by returning from a run-method (rule RETURN_{run} of Table 3).⁶

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.

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

⁶The worked-off local configuration (α, τ, ϵ) is kept in the global configuration to ensure that the thread of α cannot be started twice.

3.3.1. Proof outlines

To 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 × Int × Thread`, storing the identities of the caller object, the calling local configuration, and the caller thread. Note that the thread identities of caller and callee are the same in all cases but the invocation of a start-method. The run-method of the initial object is executed with the parameters `(thread, caller)` having the values $(\alpha_0, (null, 0, null))$, where α_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 $x := \text{if } e_0 = \text{this} \wedge \neg \text{started} \text{ then } x \text{ else } e \text{ fi}$.

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

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 $\text{caller} = (\text{this}, \text{conf}', \text{thread}')$ 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 $\text{waits_for_ret}(q, \vec{y} := \vec{e})$ replaced by

$$\text{interleavable}(q, \vec{y} := \vec{e}) \stackrel{\text{def}}{=} \text{thread} = \text{thread}' \rightarrow \text{waits_for_ret}(q, \vec{y} := \vec{e}) \wedge \\ \text{thread} \neq \text{thread}' \rightarrow \neg \text{self_start}(q, \vec{y} := \vec{e}).$$

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. $\text{CALL}_{\text{start}}$: 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 $\{q_2\}^{\text{call}} \langle \vec{y}_2 := \vec{e}_2 \rangle^{\text{call}} \{q_3\} \text{stm}; \text{return}$ is the body of the run-method of c' having formal parameters \vec{u} , and local variables \vec{v} except the formal parameters. The callee class invariant is $q_1 = I_{c'}$. Furthermore, f_{comm} is $\vec{u}', \vec{v}' := \vec{E}(z), \text{lnit}(\vec{v}), f_{\text{obs1}}$ is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{obs2} is $z'.\vec{y}_2 := \vec{E}'_2(z')$.
2. $\text{CALL}_{\text{start}}^{\text{skip}}$: For the above statements, the equations must additionally hold with the assertion comm given by $E_0(z) = z' \wedge z'.\text{started}$, $q_2 = q_3 = \text{true}$, q_1 and f_{obs1} as above, and f_{comm} and f_{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 $\text{Java}_{\text{synchron}}$, 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.

$modif$	$::=$	$nsync \mid sync$
$meth$	$::=$	$modif m(u, \dots, u) \{ stm; return \exp_{ret} \}$
$meth_{run}$	$::=$	$nsync \text{ run}() \{ stm; return \}$
$meth_{wait}$	$::=$	$nsync \text{ wait}() \{ ?signal; return_{getlock} \}$
$meth_{notify}$	$::=$	$nsync \text{ notify}() \{ !signal; return \}$
$meth_{notifyAll}$	$::=$	$nsync \text{ notifyAll}() \{ !signal_all; return \}$
$meth_{predef}$	$::=$	$meth_{start} \ meth_{wait} \ meth_{notify} \ meth_{notifyAll}$
$class$	$::=$	$class \ c \{ meth. \dots meth \ meth_{run} \ meth_{predef} \}$
$class_{main}$	$::=$	$class$
$prog$	$::=$	$\langle class \dots class \ class_{main} \rangle$

Table 9. $Java_{synchron}$ abstract syntax

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 $modif$ 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 synchronized. Furthermore, we consider the additional predefined methods `wait`, `notify`, and `notifyAll`, whose definitions use the auxiliary statements `!signal`, `!signal_all`, `?signal`, and `returngetlock`.⁸

4.2. Semantics

The operational semantics extends the semantics of $Java_{conc}$ by the rules of Table 10, where the `CALL` rule is replaced. For synchronized method calls, the lock of the callee object has to be free or owned by the executing thread, as expressed by the predicate $owns$, defined below.

The remaining rules handle the semantics of the monitor methods `wait`, `notify`, and `notifyAll`. In all three cases the caller must own the lock of the callee object (rule `CALLmonitor`). A thread can block itself on an object whose lock it owns by invoking the object’s `wait`-method, thereby relinquishing the lock and placing itself into the object’s wait set. Formally, the wait set $wait(T, \alpha)$ of an object is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, ?signal; stm)$. After having put itself on ice, the thread awaits notification by another thread which invokes the `notify`-method of the object. The `!signal` statement in the `notify`-method thus reactivates a non-deterministically chosen single thread waiting for notification on the given object (rule `SIGNAL`). Analogously to the wait set, the notified set $notified(T, \alpha)$ of α is the set of all stacks in T with top element of the form $(\alpha, \tau, return_{getlock})$, i.e., threads which have been notified and are trying to get hold of the lock again. According to rule `RETURNwait`, the receiver can continue after notification in executing `returngetlock` 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 *signal-and-continue* monitor discipline [10]. If no threads are waiting on the object, the `!signal` of the

⁷Java does not have the “non-synchronized” modifier: methods are non-synchronized by default.

⁸Java’s `Thread` class additionally support methods for suspending, resuming, and stopping a thread, but they are deprecated and thus not considered here.

$$\begin{array}{c}
\frac{m \notin \{\text{start, run, wait, notify, notifyAll}\} \quad \text{modif } m(\vec{u})\{ \text{body} \} \in \text{Meth}_c}{\frac{\beta = \llbracket e_0 \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \in \text{Val}^c(\sigma) \quad \tau' = \tau_{m,c}^{\text{init}}[\vec{u} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau}] \quad (\text{modif} = \text{sync}) \rightarrow \neg \text{owns}(T, \beta)}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, u := e_0.m(\vec{e}); \text{stm}) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{receive } u; \text{stm}) \circ (\beta, \tau', \text{body}) \}, \sigma \rangle} \text{CALL}}
\\
\frac{m \in \{\text{wait, notify, notifyAll}\}}{\frac{\beta = \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \in \text{Val}^c(\sigma) \quad \text{owns}(\xi \circ (\alpha, \tau, e.m()); \text{stm}), \beta)}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, e.m()); \text{stm} \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{receive}; \text{stm}) \circ (\beta, \tau_{m,c}^{\text{init}}, \text{body}_{m,c}) \}, \sigma \rangle} \text{CALL}_{\text{monitor}}}}
\\
\frac{\neg \text{owns}(T, \beta)}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{receive}; \text{stm}) \circ (\beta, \tau', \text{return}_{\text{getlock}}) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{stm}) \}, \sigma \rangle} \text{RETURN}_{\text{wait}}}}
\\
\frac{}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{!signal}; \text{stm}) \} \dot{\cup} \{ \xi' \circ (\alpha, \tau', \text{?signal}; \text{stm}') \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{stm}) \} \dot{\cup} \{ \xi' \circ (\alpha, \tau', \text{stm}') \}, \sigma \rangle} \text{SIGNAL}}
\\
\frac{\text{wait}(T, \alpha) = \emptyset}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{!signal}; \text{stm}) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{stm}) \}, \sigma \rangle} \text{SIGNAL}_{\text{skip}}}}
\\
\frac{T' = \text{signal}(T, \alpha)}{\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{!signal}_{\text{all}}; \text{stm}) \}, \sigma \rangle \longrightarrow \langle T' \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{stm}) \}, \sigma \rangle} \text{SIGNAL}_{\text{ALL}}}
\end{array}$$

Table 10. $\text{Java}_{\text{synchron}}$ Operational semantics

notifier is without effect (rule $\text{SIGNAL}_{\text{skip}}$). The notifyAll-method generalizes notify in that all waiting threads are notified via the !signal_all-broadcast (rule $\text{SIGNAL}_{\text{ALL}}$). The effect of this statement is given by defining $\text{signal}(T, \alpha)$ as $(T \setminus \text{wait}(T, \alpha)) \cup \{ \xi \circ (\beta, \tau, \text{stm}) \mid \xi \circ (\beta, \tau, \text{?signal}; \text{stm}) \in \text{wait}(T, \alpha) \}$.

Using the wait and notified sets, we can now formalize the *owns* predicate: A thread ξ owns the lock of β iff ξ executes some synchronized method of β , but not its wait-method. Formally, $\text{owns}(T, \beta)$ is true iff there exists a thread $\xi \in T$ and a $(\beta, \tau, \text{stm}) \in \xi$ with stm synchronized and $\xi \notin \text{wait}(T, \beta) \cup \text{notified}(T, \beta)$. 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 $\text{Java}_{\text{conc}}$, 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 × 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 × 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 `lock := inc(lock)` and `lock := dec(lock)`, respectively. The semantics of incrementing the lock $\llbracket \text{inc}(\text{lock}) \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}$ is $(\tau(\text{thread}), n+1)$ for $\sigma_{inst}(\text{lock}) = (\alpha, n)$. Decrementing `dec(lock)` is inverse: $\llbracket \text{dec}(\text{lock}) \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}$ with $\sigma_{inst}(\text{lock}) = (\alpha, n)$ 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 `?signal`, `!signal`, and `!signal_all` statements get replaced by auxiliary assignments⁹ Entering the `wait`-method gets the observation `wait, lock := wait ∪ {lock}`, `free`; returning from the `wait`-method observes `lock, notified := get(notified, thread), notified \ {get(notified, thread)}`. For a thread $\alpha ∈ Val^{\text{Thread}}$ and a list $notified ∈ Val^{\text{list}(\text{Thread} \times \text{Int})}$, `get(notified, α)` retrieves the value (α, n) from the list. The semantics assures uniqueness of the association. The `!signal` statement of the `notify`-method is represented by the auxiliary 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 `!signal_all` statement of the `notifyAll`-method is represented by the auxiliary assignment `notified, wait := notified ∪ wait, ∅`.

4.3.2. Verification conditions

Initial and local correctness agree with those for `Javaconc`. 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

⁹In `Java`, the implementation of the monitor methods are syntactically not included in class definitions. Their augmentation and annotation can be specified by special comments.

¹⁰This condition is not necessary for a minimal proof system, but reduces the number of verification conditions.

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 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.

The cooperation test We extend the cooperation test for $Java_{conc}$ 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 $z'.lock = free \vee thread(z'.lock) = thread$, where $thread$ is the caller thread, z' is the callee object, and where $thread(z'.lock)$ 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 $\{p_1\} u_{ret} := e_0.m(\vec{e}) \{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_3\}^{wait}$ (or such without receiving a value) in class c with e_0 of type c' , where method $m \notin \{\text{start, wait, notify, notifyAll}\}$ of c' is synchronized with body $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\} stm; \text{return } e_{ret}$, formal parameters \vec{u} , and local variables \vec{v} except the formal parameters. The callee class invariant is $q_1 = I_{c'}$. The assertion $comm$ is given by $E_0(z) = z' \wedge (z'.lock = free \vee thread(z'.lock) = thread)$. Furthermore, f_{comm} is $\vec{u}', \vec{v}' := \vec{E}(z), \text{Init}(\vec{v})$, f_{obs1} is given by $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{obs2} is $z'.\vec{y}_2 := \vec{E}_2(z')$. If m is not synchronized, $z'.lock = free \vee thread(z'.lock) = thread$ in $comm$ is dropped.
2. **CALL_{monitor}**: For $m \in \{\text{wait, notify, notifyAll}\}$, $comm$ is $E_0(z) = z' \wedge thread(z'.lock) = thread$.
3. **RETURN_{wait}**: For $\{q_1\} \text{return}_{getlock} \{q_2\}^{!ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret} \{q_3\}$ in a wait-method, $comm$ is $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge z'.lock = free \wedge thread' \in z'.notified$.

5. Exception handling

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

Note that concurrency and exception handling are quite orthogonal features. From the proof-theoretic point of view, the only relation between them is that exception handling occurs in a concurrent setting, i.e., interleaved. However, this kind of interleaving does not have to be distinguished from interleaving in other statements, that is mirrored in the absence of a special interference freedom test for exception handling.

Of course, from the programming side, a thread can cause another thread to throw an exception, or it can communicate with other threads during exception handling, but in the proof theory these are no special exception-handling cases, but are covered by the previous rules.

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.

$$\begin{array}{l}
 stm ::= x := e \mid u := e \mid u := \text{new}^c \\
 \quad \mid u := e.m(e, \dots, e) \mid e.m(e, \dots, e) \\
 \quad \mid \text{throw } e \mid \text{try } stm \text{ catch } (cu) \text{ } stm \dots \text{ catch } (cu) \text{ } stm \text{ finally } stm \text{ yrt} \\
 \quad \mid \epsilon \mid stm; stm \mid \text{if } e \text{ then } stm \text{ else } stm \text{ fi} \mid \text{while } e \text{ do } stm \text{ od } \dots
 \end{array}$$

Table 11. $Java_{exc}$ abstract syntax

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 arbitrary¹¹ type. The empty reference cannot be thrown.¹² 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 $Java_{synchron}$ 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 $(c_1 u_1) stm_1 \dots$; 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 $(c_1 u_1) stm_1 \dots$; catch $(c_n u_n) stm_n$ finally stm_{n+1} yrt` (rule TRY). We

¹¹In *Java* only objects extending `Throwable` may be thrown.

¹²In *Java*, a `NullPointerException` is thrown in this case.

$\tau' = \tau[\text{exc} \mapsto \tau(\text{exc}) \circ \text{null}]$	TRY
$\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{try } stm_0; \text{catch } (c_1 u_1) stm_1 \dots; \text{catch } (c_n u_n) stm_n \text{ finally } stm_{n+1} \text{ yrt}; stm') \}, \sigma \rangle \longrightarrow$	
$\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau', \text{try } stm_0; \text{catch } (c_1 u_1) stm_1 \dots; \text{catch } (c_n u_n) stm_n \text{ finally } stm_{n+1} \text{ yrt}; stm') \}, \sigma \rangle$	
$0 \leq n$	FINALLY
$\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{catch } (c_1 u_1) stm_1 \dots; \text{catch } (c_n u_n) stm_n \text{ finally } stm_{n+1} \text{ yrt}; stm) \}, \sigma \rangle \longrightarrow$	
$\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, stm_{n+1} \text{ yrt}; stm) \}, \sigma \rangle$	
$\tau(\text{exc}) = \beta_0 \circ \dots \circ \beta_k \circ \beta_{k+1} \quad \tau' = \tau[\text{exc} \mapsto \beta_0 \circ \dots \circ \beta_k][\text{top} \mapsto \beta_{k+1}]$	
$\text{if } \tau'(\text{top}) = \text{null} \text{ then } stm' = stm \text{ else } stm' = \text{throw top}; stm \text{ fi}$	YRT
$\langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{yrt}; stm) \}, \sigma \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau', stm') \}, \sigma \rangle$	

Table 12. $Java_{exc}$ Operational semantics (1)

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 ϵ , 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 THROW₁) and the finally clause (rule FINALLY with $n = 0$) get executed. Otherwise, if no exceptions have been thrown (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).

¹³Note that for example `catch (c2 u2) stm2` is not a statement.

$ \begin{array}{l} stm \text{ is try-closed} \quad stm' = \text{catch}(c_1 u_1) stm_1 \dots; \text{catch}(c_n u_n) stm_n \text{ finally } stm_{n+1} \text{ yrt} \\ 1 \leq i \leq n \quad \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \in Val^{c_i} \quad \forall 1 \leq j < i. \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \notin Val^{c_j} \\ \tau' = \tau[u_i \mapsto \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau}] \end{array} $	THROW ₁	
$ \begin{array}{l} \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{throw } e; stm; stm'; stm''), \sigma \} \rangle \longrightarrow \\ \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau', stm_i; \text{finally } stm_{n+1} \text{ yrt}; stm''), \sigma \} \rangle \end{array} $		
$ \begin{array}{l} stm \text{ is try-closed} \quad stm' = \text{catch}(c_1 u_1) stm_1 \dots; \text{catch}(c_n u_n) stm_n \text{ finally } stm_{n+1} \text{ yrt} \\ \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \neq null \quad 0 \leq n \quad \forall 1 \leq i \leq n. \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \notin Val^{c_i} \\ \tau(\text{exc}) = \beta_0 \circ \dots \circ \beta_k \circ \beta_{k+1} \quad \tau' = \tau[\text{exc} \mapsto \beta_0 \circ \dots \circ \beta_k \circ \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau}] \end{array} $	THROW ₂	
$ \begin{array}{l} \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{throw } e; stm; stm'; stm''), \sigma \} \rangle \longrightarrow \\ \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau', stm_{n+1} \text{ yrt}; stm''), \sigma \} \rangle \end{array} $		
$ \begin{array}{l} stm \text{ is try-closed} \\ \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \neq null \quad \tau(\text{exc}) = \beta_0 \circ \dots \circ \beta_k \circ \beta_{k+1} \quad \tau' = \tau[\text{exc} \mapsto \beta_0 \circ \dots \circ \beta_k \circ \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau}] \end{array} $		THROW ₃
$ \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{throw } e; stm \text{ yrt}; stm'), \sigma \} \rangle \longrightarrow \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau', \text{yrt}; stm'), \sigma \} \rangle $		
$ \begin{array}{l} stm' \text{ is try-closed} \quad \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\beta), \tau'} \neq null \quad \tau'' = \tau[\text{top} \mapsto \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\beta), \tau'}] \end{array} $	THROW ₄	
$ \begin{array}{l} \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau, \text{receive } u_{ret}; stm) \circ (\beta, \tau', \text{throw } e; stm'), \sigma \} \rangle \longrightarrow \\ \langle T \dot{\cup} \{ \xi \circ (\alpha, \tau'', \text{throw top}; stm), \sigma \} \rangle \end{array} $		
$ \begin{array}{l} stm \text{ is try-closed} \quad \llbracket e \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau} \neq null \\ \langle T \dot{\cup} \{ (\alpha, \tau, \text{throw } e; stm; \text{return}), \sigma \} \rangle \longrightarrow \langle T \dot{\cup} \{ (\alpha, \tau, \text{return}), \sigma \} \rangle \end{array} $		THROW ₅

Table 13. *Java_{exc}* Operational semantics (2)

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_{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 con-

struct 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

$$\{p_0\} \text{ throw } u \{p_1\}^{\text{throw}} \langle \vec{y} := \vec{e} \rangle^{\text{throw}} \{p_2\} .$$

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:

$$\begin{array}{l} \{p_0\} \quad u := e_0.m(\vec{e}) \quad \{p_1\}^{\text{!call}} \quad \langle \vec{y}_1 := \vec{e}_1 \rangle^{\text{!call}} \\ \{p_2\}^{\text{wait}} \quad \{p_3\}^{\text{?ret}} \quad \langle \vec{y}_4 := \vec{e}_4 \rangle^{\text{?ret}} \\ \{p_4\}^{\text{exc}} \quad \{p_5\}^{\text{rethrow}} \quad \langle \vec{y}_{\text{thr}} := \vec{e}_{\text{thr}} \rangle^{\text{rethrow}} \\ \{p_6\} . \end{array}$$

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 $\vec{y}_{\text{thr}} := \vec{e}_{\text{thr}}$. 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 $\vec{y}_4 := \vec{e}_4$.

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 $\vec{y}_{\text{try}} := \vec{e}_{\text{try}}$, 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 $\vec{y}_{\text{yrt}} := \vec{e}_{\text{yrt}}$ of exiting. If there is an exception to be rethrown, the assertion p_{exc} is required to hold after the observation of yrt, p_{thr} should hold after rethrowing and prior to its observation $\vec{y}_{\text{thr}} := \vec{e}_{\text{thr}}$. 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 `list Object` with initial value ϵ 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 `thrown` as a shortcut for `tail(exc) ≠ null`, where the function `tail(v)` gives the last element of the sequence v . We use also the function `head(v)` which returns the sequence v without its last element¹⁴. Note that the variables `exc` and `top` are *local*. In the concurrent setting, all threads have their own exception mechanism, which are independent of each other.

¹⁴These functions are applied to non-empty sequences only.

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 throw statement outside try-catch-finally blocks in a synchronized method by the assignment $\text{lock} := \text{dec}(\text{lock})$.

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 $\text{Java}_{\text{synchron}}$. 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.

Entering a try block pushes an empty reference on the exception stack (rule 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 has been thrown in a try or in a catch block, then after termination of the block execution continues in the finally block (rule 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 YRT) is covered by the Conditions (13)-(15). Condition (13) assures that p_{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_{exc} is required to hold after the observation of yrt 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 THROW₁ and 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 THROW₄) is covered by the conditions of the cooperation test for exception handling.

Definition 5.1. (Local correctness: Exception handling)

A proof outline is *locally correct* under exception handling, if for all statements stm of the form

$$\begin{array}{l}
\{p\} \quad \text{try} \quad \{p_{\text{try}}\}^{\text{try}} \quad \langle \vec{y}_{\text{try}} := \vec{e}_{\text{try}} \rangle^{\text{try}} \quad \{p_0\} \quad stm_{\text{try}}; \quad \{p'_0\} \\
\quad \text{catch}(c_1 u_1) \quad \{p_1\} \quad stm_1; \quad \{p'_1\} \\
\quad \dots \\
\quad \text{catch}(c_n u_n) \quad \{p_n\} \quad stm_n; \quad \{p'_n\} \\
\quad \text{finally} \quad \{p_{\text{fin}}\} \quad stm_{\text{fin}} \quad \{p'_{\text{fin}}\} \\
\quad \text{yrt} \quad \{p_{\text{yrt}}\}^{\text{yrt}} \quad \langle \vec{y}_{\text{yrt}} := \vec{e}_{\text{yrt}} \rangle^{\text{yrt}} \\
\{p_{\text{exc}}\}^{\text{exc}} \quad \{p_{\text{thr}}\}^{\text{rethrow}} \quad \langle \vec{y}_{\text{thr}} := \vec{e}_{\text{thr}} \rangle^{\text{rethrow}} \\
\{p'\} ,
\end{array}$$

and for all $0 \leq i \leq n$,

$$\models_{\mathcal{L}} \{p\} \text{ exc} := \text{exc} \circ \text{null} \{p_{\text{try}}\} , \quad (10)$$

$$\models_{\mathcal{L}} \{p\} \text{ exc} := \text{exc} \circ \text{null}; \vec{y}_{\text{try}} := \vec{e}_{\text{try}} \{p_0\} , \quad (11)$$

$$\models_{\mathcal{L}} p'_i \rightarrow p_{\text{fin}} , \quad (12)$$

$$\models_{\mathcal{L}} \{p'_{\text{fin}}\} \text{ exc}, \text{top} := \text{head}(\text{exc}), \text{tail}(\text{exc}) \{p_{\text{yrt}}\} , \quad (13)$$

$$\models_{\mathcal{L}} \{p'_{\text{fin}} \wedge \text{tail}(\text{exc}) = \text{null}\} \text{ exc}, \text{top} := \text{head}(\text{exc}), \text{tail}(\text{exc}); \vec{y}_{\text{yrt}} := \vec{e}_{\text{yrt}} \{p'\} , \quad (14)$$

$$\models_{\mathcal{L}} \{p'_{\text{fin}} \wedge \text{tail}(\text{exc}) \neq \text{null}\} \text{ exc}, \text{top} := \text{head}(\text{exc}), \text{tail}(\text{exc}); \vec{y}_{\text{yrt}} := \vec{e}_{\text{yrt}} \{p_{\text{exc}}\} , \quad (15)$$

and for all statements $\{q_0\} \text{ throw } e \{q_1\}^{\text{throw}} \langle \vec{y} := \vec{e} \rangle^{\text{throw}}$ in stm_{try} which do not occur in an inner try-catch-finally block inside stm_{try} , and for all $1 \leq i \leq n$,

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \text{null} \wedge \text{hastype}(e, c_i) \wedge \forall 1 \leq j < i. \neg \text{hastype}(e, c_j)\} u_i := e \{q_1\} \quad (16)$$

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \text{null} \wedge \text{hastype}(e, c_i) \wedge \forall 1 \leq j < i. \neg \text{hastype}(e, c_j)\} u_i := e; \vec{y} := \vec{e} \{p_i\} \quad (17)$$

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \text{null} \wedge \forall 1 \leq j \leq n. \neg \text{hastype}(e, c_j)\} \text{ exc} := \text{head}(\text{exc}) \circ e \{q_1\} \quad (18)$$

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \text{null} \wedge \forall 1 \leq j \leq n. \neg \text{hastype}(e, c_j)\} \text{ exc} := \text{head}(\text{exc}) \circ e; \vec{y} := \vec{e} \{p_{\text{fin}}\} . \quad (19)$$

For statements $\{q_0\} \text{ throw } e \{q_1\}^{\text{throw}} \langle \vec{y} := \vec{e} \rangle^{\text{throw}}$ in catch blocks, (18) and (19) are required to hold without the antecedent $\forall 1 \leq j \leq n. \neg \text{hastype}(e, c_j)$. For throw statements in finally blocks, (18) and (19) should hold without the above antecedent and with p_{fin} replaced by p'_{fin} . The above conditions (16)-(19) should hold also for statements of the form $\{q_0\}^{\text{exc}} \{q_1\}^{\text{rethrow}} \langle \vec{y} := \vec{e} \rangle^{\text{rethrow}}$, where the expression e in the conditions is replaced by top . Finally, for statements of the form $\{q_0\} \text{ throw } e \{q_1\}^{\text{throw}} \langle \vec{y} := \vec{e} \rangle^{\text{throw}}$ outside try-catch-finally blocks in a run-method with body stm' ; return , (18) and (19) should hold without the above antecedent, with p_{fin} replaced by $\text{pre}(\text{return})$, and without the update of exc . The above conditions must hold also for all statements $\{q_0\} \{q_1\}^{\text{rethrow}} \langle \vec{y} := \vec{e} \rangle^{\text{rethrow}}$, 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 $\text{Java}_{\text{synchron}}$ 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: $u_{ret} := e_0.m(\vec{e}) \dots \{p_1\}^{wait} \quad \{p_2\}^{?ret} \quad \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} \quad \{p_3\}^{exc} \dots$
 callee: $\dots \{q_1\} \quad \text{throw } e \quad \{q_2\}^{throw} \quad \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} \dots$

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 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 $z'.\vec{y}'_3 := \vec{E}'_3(z')$ and $z.\vec{y}_4 := \vec{E}_4(z)$, respectively. Note that if the invoked method is synchronized, than the observation $z'.\vec{y}'_3 := \vec{E}'_3(z')$ decrements the value of the lock of z' by the built-in augmentation.

We use the assertion `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 $e_0.m(\vec{e})$. 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 $\vec{u}' = \vec{E}(z)$ states that the values of the formal and of the actual parameters agree, which implies that the primed built-in auxiliary formal parameter `caller'` of the callee stores $(z, \text{conf}, \text{thread})$ identifying the caller. I.e., the assertion $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$ 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 $u_{ret} := e_0.m(\vec{e}) \langle \text{stm} \rangle^{call} \{p_1\}^{wait} \{p_2\}^{?ret} \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} \{p_3\}^{exc}$ (or such without receiving a value) occurring in class c with $m \neq \text{start}$ and e_0 of type c' , and for all $\{q_1\} \text{ throw } e \{q_2\}^{throw} \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw}$ in $m(\vec{u})$ of c' which are not inside any try-catch-finally statement,

$$\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q'_1(z') \wedge \text{comm}\} \quad \text{top} := E'(z') \quad \{P_2(z) \wedge Q'_2(z')\} \quad \text{and} \quad (20)$$

$$\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q'_1(z') \wedge \text{comm}\} \quad \text{top} := E'(z'); z'.\vec{y}'_3 := \vec{E}'_3(z'); z.\vec{y}_4 := \vec{E}_4(z) \quad (21)$$

$$\{GI \wedge P_3(z)\}$$

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

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

6. Weakest precondition calculus

The verification conditions of the previous sections were formulated as standard Hoare-triples. In this section we define their formal semantics, given by means of a weakest precondition calculus. To do so, first we introduce substitutions in Section 6.1, before re-formulating the verification conditions for $Java_{exc}$ in Section 6.2 to logical implications, using the substitutions.

6.1. Substitution operations

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

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

Lemma 6.1. (Local substitution)

For arbitrary logical environments ω and instance local states (σ_{inst}, τ) we have

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p[\vec{e}/\vec{y}] \quad \text{iff} \quad \omega, \sigma_{inst}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{L}}^{\omega, \sigma_{inst}, \tau}], \tau[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{L}}^{\omega, \sigma_{inst}, \tau}] \models_{\mathcal{L}} p.$$

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

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

The substitution is extended to global assertions homomorphically. We will also use the substitution $P[\vec{E}/z.\vec{y}]$ for arbitrary variable sequences \vec{y} possibly containing logical variables, whose semantics is defined by the simultaneous substitutions $[\vec{E}_x/z.\vec{x}]$ and $[\vec{E}_u/\vec{u}]$, where \vec{x} and \vec{u} are the sequences of the instance and logical variables¹⁵ of \vec{y} , and \vec{E}_x and \vec{E}_u the corresponding subsequences of \vec{E} and

¹⁵Local variables are viewed as logical ones in the global assertion language.

$[\vec{E}_u/\vec{u}]$ is the usual capture-avoiding substitution like in the local substitution; if only logical variables are substituted, we simply write $P[\vec{E}/\vec{u}]$. That the substitution accurately catches the semantical update, and thus represents the weakest precondition relation, is expressed by the following lemma:

Lemma 6.2. (Global substitution)

For arbitrary global states σ and logical environments ω referring only to values existing in σ we have

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

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

6.2. Verification conditions

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

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

Definition 6.1. (Initial correctness)

A proof outline is *initially correct*, if

$$\begin{aligned} \models_{\mathcal{G}} \text{InitState}(z) \wedge (\forall z'. z' = \text{null} \vee z = z') \rightarrow \\ P_2(z) \circ f_{init} \wedge (GI \wedge P_3(z) \wedge I_c(z)) \circ f_{obs} \circ f_{init}, \end{aligned} \quad (22)$$

where c is the main class, $\{p_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{p_3\} \text{stm}; \text{return}$ is the body and \vec{v} the local variables of the run-method of c , $z \in LVar^c$, and $z' \in LVar^{\text{Object}}$. Furthermore,

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

Definition 6.2. (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 , being an unobserved assignment, an alone-standing observation, or an observed assignment,

$$\models_{\mathcal{L}} p_1 \rightarrow p_2 \circ f_{ass}, \quad (23)$$

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

Definition 6.3. (Local correctness: Exception handling)

A proof outline is *locally correct* under exception handling, if for all statements stm of the form

$$\begin{array}{llll}
\{p\} & \text{try} & \{p_{\text{try}}\}^{\text{try}} & \langle \vec{y}_{\text{try}} := \vec{e}_{\text{try}} \rangle^{\text{try}} & \{p_0\} & stm_{\text{try}}; & \{p'_0\} \\
& \text{catch}(c_1 u_1) & & & \{p_1\} & stm_1; & \{p'_1\} \\
& \dots & & & & & \\
& \text{catch}(c_n u_n) & & & \{p_n\} & stm_n; & \{p'_n\} \\
& \text{finally} & & & \{p_{\text{fin}}\} & stm_{\text{fin}} & \{p'_{\text{fin}}\} \\
\{p_{\text{exc}}\}^{\text{exc}} & \text{yrt} & \{p_{\text{yrt}}\}^{\text{yrt}} & \langle \vec{y}_{\text{yrt}} := \vec{e}_{\text{yrt}} \rangle^{\text{yrt}} & & & \\
\{p'\} & & \{p_{\text{thr}}\}^{\text{rethrow}} & \langle \vec{y}_{\text{throw}} := \vec{e}_{\text{throw}} \rangle^{\text{rethrow}} & & &
\end{array}$$

and for all $0 \leq i \leq n$,

$$\models_{\mathcal{L}} p \rightarrow p_{\text{try}}[\text{exc} \circ \text{null}/\text{exc}] \wedge p_0[\vec{e}_{\text{try}}/\vec{y}_{\text{try}}][\text{exc} \circ \text{null}/\text{exc}], \quad (24)$$

$$\models_{\mathcal{L}} p'_i \rightarrow p_{\text{fin}}, \quad (25)$$

$$\models_{\mathcal{L}} p'_{\text{fin}} \rightarrow p_{\text{yrt}}[\text{head}(\text{exc}), \text{tail}(\text{exc})/\text{exc}, \text{top}], \quad (26)$$

$$\models_{\mathcal{L}} (p'_{\text{fin}} \wedge \text{tail}(\text{exc}) = \text{null}) \rightarrow p'[\vec{e}_{\text{yrt}}/\vec{y}_{\text{yrt}}][\text{head}(\text{exc}), \text{tail}(\text{exc})/\text{exc}, \text{top}], \quad (27)$$

$$\models_{\mathcal{L}} (p'_{\text{fin}} \wedge \text{tail}(\text{exc}) \neq \text{null}) \rightarrow p_{\text{exc}}[\vec{e}_{\text{yrt}}/\vec{y}_{\text{yrt}}][\text{head}(\text{exc}), \text{tail}(\text{exc})/\text{exc}, \text{top}], \quad (28)$$

and for all statements $\{q_0\} \text{ throw } e \{q_1\}^{\text{throw}} \langle \vec{y} := \vec{e} \rangle^{\text{throw}}$ in stm_{try} which does not occur in an inner try-catch-finally block inside stm_{try} , and for all $1 \leq i \leq n$,

$$\models_{\mathcal{L}} (q_0 \wedge e \neq \text{null} \wedge \text{hastype}(e, c_i) \wedge \forall 1 \leq j < i. \neg \text{hastype}(e, c_j)) \rightarrow q_1[e/u_i] \wedge p_i[\vec{e}/\vec{y}][e/u_i], \quad (29)$$

$$\models_{\mathcal{L}} (q_0 \wedge e \neq \text{null} \wedge \forall 1 \leq j \leq n. \neg \text{hastype}(e, c_j)) \rightarrow q_1[\text{head}(\text{exc}) \circ e/\text{exc}] \wedge p_{\text{fin}}[\vec{e}/\vec{y}][\text{head}(\text{exc}) \circ e/\text{exc}]. \quad (30)$$

For statements $\{q_0\} \text{ throw } e \{q_1\}^{\text{throw}} \langle \vec{y} := \vec{e} \rangle^{\text{throw}}$ in catch blocks, (30) is required to hold without the antecedent $\forall 1 \leq j \leq n. \neg \text{hastype}(e, c_j)$. For throw statements in finally blocks, (30) should hold without the above antecedent and with p_{fin} replaced by p'_{fin} . The above conditions (29) and (30) should hold also for statements of the form $\{q_0\} \{q_1\}^{\text{rethrow}} \langle \vec{y} := \vec{e} \rangle^{\text{rethrow}}$, where the expression e in the conditions is replaced by top. Finally, for statements of the form $\{q_0\} \text{ throw } e \{q_1\}^{\text{throw}} \langle \vec{y} := \vec{e} \rangle^{\text{throw}}$ outside try-catch-finally blocks in a run-method with body stm' ; return, Condition (30) should hold without the above antecedent, without the update of exc, and with the assertion p_{fin} replaced by $pre(\text{return})$. For statements $\{q_0\} \{q_1\}^{\text{rethrow}} \langle \vec{y} := \vec{e} \rangle^{\text{rethrow}}$ the same conditions must hold where we additionally replace e by top.

Definition 6.4. (Interference freedom)

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

$$\models_{\mathcal{L}} p \wedge I_c \rightarrow I_c \circ f_{\text{ass}}, \quad (31)$$

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

$$\models_{\mathcal{L}} p \wedge q' \wedge \text{interleavable}(q, \vec{y} := \vec{e}) \rightarrow q' \circ f_{ass} . \quad (32)$$

Definition 6.5. (Cooperation test: Communication)

A proof outline satisfies the *cooperation test for communication*, if

$$\begin{aligned} \models_{\mathcal{G}} \quad & GI \wedge P_1(z) \wedge Q'_1(z') \wedge \text{comm} \wedge z \neq \text{null} \wedge z' \neq \text{null} \rightarrow \\ & (P_2(z) \wedge Q'_2(z')) \circ f_{comm} \wedge \\ & (GI \wedge P_3(z) \wedge Q'_3(z')) \circ f_{obs2} \circ f_{obs1} \circ f_{comm} \end{aligned} \quad (33)$$

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

- 1 (a) **CALL**: For all calls $\{p_1\} u_{ret} := e_0.m(\vec{e})\{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_3\}^{wait}$ (or such without receiving a value) in class c with e_0 of type c' , where method $m \notin \{\text{start}, \text{wait}, \text{notify}, \text{notifyAll}\}$ of c' is synchronized with body $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\} stm; \text{return } e_{ret}$, formal parameters \vec{u} , and local variables \vec{v} except the formal parameters. The callee class invariant is $q_1 = I_{c'}$. The assertion comm is given by $E_0(z) = z' \wedge (z'.\text{lock} = \text{free} \vee \text{thread}(z'.\text{lock}) = \text{thread})$. Furthermore, $f_{comm} = [\vec{E}(z), \text{Init}(\vec{v})/\vec{u}', \vec{v}']$, $f_{obs1} = [\vec{E}_1(z)/z.\vec{y}_1]$, $f_{obs2} = [\vec{E}_2(z')/z'.\vec{y}_2]$. If m is not synchronized, $z'.\text{lock} = \text{free} \vee \text{thread}(z'.\text{lock}) = \text{thread}$ in comm is dropped.
 - (b) **CALL_{monitor}**: For $m \in \{\text{wait}, \text{notify}, \text{notifyAll}\}$, comm is given by $E_0(z) = z' \wedge \text{thread}(z'.\text{lock}) = \text{thread}$.
 - (c) **CALL_{start}**: For $m = \text{start}$, comm is $E_0(z) = z' \wedge \neg z'.\text{started}$, where $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\} stm; \text{return}$ is the body of the run-method of c' .
 - (d) **CALL_{start}^{skip}**: For $m = \text{start}$, additionally, (33) must hold with comm given by $E_0(z) = z' \wedge z'.\text{started}$, $q_2 = q_3 = \text{true}$, and f_{comm} and f_{obs2} are the identity functions.
- 2 (a) **RETURN**: For all method call statements $u_{ret} := e_0.m(\vec{e}) \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_1\}^{wait} \{p_2\}^{?ret} \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} \{p_3\}$ (or such without receiving a value) occurring in c with e_0 of type c' , such that method $m(\vec{u})$ of c' has the return statement $\{q_1\} \text{return } e_{ret} \{q_2\}^{!ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret} \{q_3\}$, Equation (33) must hold with comm given by $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$, and where $f_{comm} = [E'_{ret}(z')/u_{ret}]$, $f_{obs1} = [\vec{E}'_3(z')/z'.\vec{y}_3]$, and $f_{obs2} = [\vec{E}_4(z)/z.\vec{y}_4][\text{null}/\text{top}]$.
 - (b) **RETURN_{wait}**: For $\{q_1\} \text{return}_{getlock} \{q_2\}^{!ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret} \{q_3\}$ in a wait-method, comm is $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge z'.\text{lock} = \text{free} \wedge \text{thread}' \in z'.\text{notified}$.
 - (c) **RETURN_{run}**: For $\{q_1\} \text{return} \{q_2\}^{!ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret} \{q_3\}$ occurring in a run-method, $p_1 = p_2 = p_3 = \text{true}$, $\text{comm} = \text{true}$, and furthermore f_{comm} and f_{obs2} the identity function.

Definition 6.6. (Cooperation test: Instantiation)

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

$$\begin{aligned} \models_{\mathcal{G}} \quad & z \neq \text{null} \wedge z \neq u \wedge \exists z'. (\text{Fresh}(z', u) \wedge (GI \wedge \exists u. P_1(z)) \downarrow z') \rightarrow \\ & P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z)) \circ f_{obs} , \end{aligned} \quad (34)$$

with $z \in LVar^{c'}$ and $z' \in LVar^{\text{listObject}}$ fresh, and where $f_{obs} = [\vec{E}(z)/z.\vec{y}]$.

Definition 6.7. (Cooperation test: Exception handling)

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

$$\begin{aligned} \models_{\mathcal{G}} \quad & GI \wedge P_1(z) \wedge Q'_1(z') \wedge \text{comm} \\ & \rightarrow (P_2(z) \wedge Q'_2(z')) \circ f_{throw} \wedge (GI \wedge P_3(z)) \circ f_{obs2} \circ f_{obs1} \circ f_{throw} \end{aligned}$$

must hold with distinct fresh logical variables $z \in LVar^c$ and $z' \in LVar^{c'}$, and with comm given by $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge E'(z') \neq \text{null} \wedge z \neq \text{null} \wedge z' \neq \text{null}$. Furthermore, f_{throw} is $[E'(z')/\text{top}]$, f_{obs1} is $[\vec{E}'_3(z')/z'.\vec{y}_3]$, and f_{obs2} is $[\vec{E}_4(z)/z.\vec{y}_4]$. Rethrowing outside try-catch-finally blocks in run methods is similar.

7. 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 $prog$, we will write φ_{prog} or just φ for its annotation, and write $prog \models \varphi$, if $prog$ satisfies all requirements stated in the assertions, and $prog' \vdash \varphi'$, if $prog'$ with annotation φ' satisfies the verification conditions of the proof system:

Definition 7.1. Given a program $prog$ with annotation φ , then $prog \models \varphi$ iff for all reachable configurations $\langle T, \sigma \rangle$ of $prog$, for all $(\alpha, \tau, stm) \in T$, and for all logical environments ω referring only to values existing in σ :

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

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

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

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

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

7.1. 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 7.1. (Soundness)

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

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

The soundness proof is basically an induction on the length of computation, simultaneously on all three parts from Definition 7.1. 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.

7.2. Completeness

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

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

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

1. augment each program with enough information (see Definition 7.2 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 7.3 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 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 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 $\bigcup_{c \in \mathcal{C}} \{\text{new}^c\} \cup \bigcup_{m \in \mathcal{M}} \{!m, ?m\} \cup \{!return, ?return, !throw, ?throw\}$ of constants for this purpose, where \mathcal{C} and \mathcal{M} are the sets of all class and method names, respectively. Notification does not update the communication history, since it is object-internal computation. For the same reason, we 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 [4] for such a compositional semantics.

Definition 7.2. (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

$$h_{\text{inst}} := h_{\text{inst}} \circ ((\vec{x}, \vec{v})[\vec{e}/\vec{y}]) ,$$

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 $u_{\text{ret}} := e_0.m(\vec{e}')$ and of the corresponding reception of the return value get extended with the assignment

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

instead, if $m \neq \text{start}$. For $e_0.\text{start}(\vec{e}'); \langle \vec{y} := \vec{e} \rangle^{\text{call}}$ 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 run gets extended by

$$h_{comm} := \text{if } (\text{partner} = \text{this}) \text{ then } h_{comm} \text{ else } h_{comm} \circ (\text{sender}, \text{receiver}, \text{values}) \text{ fi},$$

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

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

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

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

Next we introduce the annotation for the augmented program.

Definition 7.3. (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 $Val(\sigma) = Val(\sigma')$, and for all $\alpha \in Val(\sigma)$, $\sigma(\alpha)(h_{comm}) = \sigma'(\alpha)(h_{comm})$.
2. For each class c , let $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} I_c$ iff there is a reachable $\langle T, \sigma \rangle$ such that $\sigma(\alpha) = \sigma_{inst}$, where $\alpha = \sigma_{inst}(\text{this})$. 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, $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(stm)$ iff there is a reachable $\langle T, \sigma \rangle$ with $\sigma(\alpha) = \sigma_{inst}$ for $\alpha = \sigma_{inst}(\text{this})$, and $(\alpha, \tau, stm; stm') \in T$.
4. For preconditions p of observations observing a statement stm which is not an assignment, let $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ iff there is a reachable $\langle T, \sigma \rangle$ with $\sigma(\alpha) = \sigma_{inst}$ for $\alpha = \sigma_{inst}(\text{this})$, and with $(\alpha, \tau', stm; stm') \in T$ enabled to execute resulting in the local state τ 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 $(\alpha, \tau', stm; stm') \in T$, we require that a call of the method of α is enabled in $\langle T, \sigma \rangle$ with resulting callee local state τ directly after communication¹⁶.

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 $prog'$.

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

Completeness for the interference freedom test and the cooperation test are more complex, since, unlike initial and local correctness, the verification conditions in these cases mention more than one local configuration in their respective antecedents. Now, the reachability assertions of $prog'$ guarantee that, when satisfied by an instance local state, there *exists* a reachable global configuration responsible for the satisfaction. So a crucial step in the completeness proof for interference freedom and the cooperation test is to show that individual reachability of two local configurations 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 7.1. (Local merging lemma)

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

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

Lemma 7.2. (Global merging lemma)

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

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

This brings us to the completeness result:

Theorem 7.2. (Completeness)

For a program $prog$, the proof outline $prog'$ satisfies the verification conditions of the proof system from Section 5.3.

¹⁶For the precondition of the observation stm at the beginning of the run-method of the main class, $\langle T, \sigma \rangle$ can also be the initial configuration before the execution of the observation stm .

8. 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. Currently, the proof system is not compositional in that it contains no explicit composition rules for combining proofs of “sub-systems” into a proof of a composed system. However, the assertions used in the proof system follow the structure of the language already—we called it modular—in that it is presented at three different levels of the language: at the global level, at the level of the classes, and finally at the local level inside single methods. Related to this separation of concerns, the proof obligations are cleanly split into local proof obligations, those dealing with interference within objects, and the cooperation tests at the global level. Indeed, adding appropriate auxiliary variables or observations, when developing the modular proof system, corresponds basically in developing a compositional *semantics* of the behavior of the language. Furthermore, the proof of soundness and especially completeness of the modular proof system shows that semantics encoded in the augmentation, in particular, in the history variables, indeed is compositional (even if not necessarily fully abstract. See e.g., [55] for a fully abstract trace semantics for class-based object-oriented programs). We see this as a benefit of undertaking the effort to provide a sound and complete modular proof system: it paves the way towards a truly compositional proof system, i.e., one that not just embodies a compositional semantics in the augmentations, but additionally has compositional *proof rules*.

Though the proof method is tool-supported, the annotation and augmentation must be given by the user. This task can be very complex, making the application difficult in practice. To increase practical relevance, an additional tool support for annotation generation would be of great benefit.

References

- [1] Abraham, E.: *An Assertional Proof System for Multithreaded Java — Theory and Tool Support*, Ph.D. Thesis, University of Leiden, 2004, Defended 20.1.2005.
- [2] Abraham, E., de Boer, F. S., de Roever, W.-P., Steffen, M.: Inductive Proof-Outlines for Monitors in Java, in: Najm et al. [34], 155–169, A longer version appeared as technical report TR-ST-03-1, April 2003.
- [3] Abraham, E., de Boer, F. S., de Roever, W.-P., Steffen, M.: *Inductive Proof Outlines for Multithreaded Java with Exceptions*, Technical Report 0313, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, December 2003.

- [4] Ábrahám, E., de Boer, F. S., de Roever, W.-P., Steffen, M.: A Compositional Operational Semantics for Java_{MT}, *International Symposium on Verification (Theory and Practice), July 2003* (N. Derschowicz, Ed.), 2772, Springer-Verlag, 2004, A preliminary version appeared as Technical Report TR-ST-02-2, May 2002.
- [5] Ábrahám, E., de Boer, F. S., de Roever, W.-P., Steffen, M.: An Assertion-Based Proof System for Multithreaded Java, *Theoretical Computer Science*, **331**, 2005.
- [6] Ábrahám-Mumm, E., de Boer, F. S.: Proof-Outlines for Threads in Java, *Proceedings of CONCUR 2000* (C. Palamidessi, Ed.), 1877, Springer-Verlag, August 2000.
- [7] Ábrahám-Mumm, E., de Boer, F. S., de Roever, W.-P., Steffen, M.: A Tool-Supported Proof System for Monitors in Java, in: Bonsangue et al. [20], 1–32.
- [8] Ábrahám-Mumm, E., de Boer, F. S., de Roever, W.-P., Steffen, M.: Verification for Java’s Reentrant Multithreading Concept, *Proceedings of FoSSaCS 2002* (M. Nielsen, U. H. Engberg, Eds.), 2303, Springer-Verlag, April 2002, A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.
- [9] Alves-Foss, J., Ed.: *Formal Syntax and Semantics of Java*, vol. 1523 of *Lecture Notes in Computer Science State-of-the-Art-Survey*, Springer-Verlag, 1999.
- [10] Andrews, G. R.: *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.
- [11] Apt, K. R.: Ten Years of Hoare’s Logic: A Survey – Part I, *ACM Transactions on Programming Languages and Systems*, **3**(4), October 1981, 431–483.
- [12] Apt, K. R., Francez, N., de Roever, W.-P.: A Proof System for Communicating Sequential Processes, *ACM Transactions on Programming Languages and Systems*, **2**, 1980, 359–385.
- [13] The Project Bali, <http://isabelle.in.tum.de/Bali/>, 2003.
- [14] van den Berg, J., Huisman, M., Jacobs, B., Poll, E.: A Type-Theoretic Memory Model for Verification of Sequential Java Programs, *Recent Trends in Algebraic Development Techniques* (D. Bert, C. Choppy, P. Mosses, Eds.), 1827, Springer-Verlag, 2000, An earlier version appeared as Computer Science Institute, University of Nijmegen, Technical Report CSI-R9926, 1999.
- [15] van den Berg, J., Jacobs, B.: The Loop Compiler for Java and JML, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’02)* (T. Margaria, W. Yi, Eds.), 2031, Springer-Verlag, 2002.
- [16] van den Berg, J., Jacobs, B., Poll, E.: Formal Specification and Verification of JavaCard’s Application Identifier Class, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France* (I. Attali, T. Jensen, Eds.), 2001.
- [17] de Boer, F. S.: A WP-Calculus for OO, *Proceedings of FoSSaCS ’99* (W. Thomas, Ed.), 1578, Springer-Verlag, 1999.
- [18] de Boer, F. S., Pierik, C.: Computer-Aided Specification and Verification of Annotated Object-Oriented Programs, *Proceedings of the Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)* (B. Jacobs, A. Rensink, Eds.), 209, Kluwer, 2002.
- [19] de Boer, F. S., Pierik, C.: *Towards an Environment for the Verification of Annotated Object-Oriented Programs*, Technical report UU-CS-2003-002, Institute of Information and Computing Sciences, University of Utrecht, January 2003.
- [20] Bonsangue, M. M., de Boer, F. S., de Roever, W.-P., Graf, S., Eds.: *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002), Leiden*, vol. 2852 of *Lecture Notes in Computer Science*, Springer-Verlag, 2003.

- [21] Cenciarelli, P., Knapp, A., Reus, B., Wirsing, M.: An Event-Based Structural Operational Semantics of Multi-Threaded Java, in: Alves-Foss [9], 157–200.
- [22] Floyd, R. W.: Assigning Meanings to Programs, *Proc. Symposia in Applied Mathematics: Mathematical Aspects of Computer Science* (J. T. Schwartz, Ed.), 1967.
- [23] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming, *Communications of the ACM*, **12**(10), 1969, 576–580.
- [24] Huisman, M.: *Java Program Verification in Higher-Order Logic with PVS and Isabelle*, Ph.D. Thesis, University of Nijmegen, 2001.
- [25] Huisman, M., Jacobs, B.: Inheritance in Higher Order Logic: Modeling and Reasoning, *Theorem Proving in Higher Order Logics (TPHOL 2000)* (M. Aagaard, J. Harrison, Eds.), 1869, An earlier version appeared as Technical Report CSI-R0004 Computing Science Institute, University of Nijmegen., 2000.
- [26] Huisman, M., Jacobs, B., van den Berg, J.: A Case Study in Class Library Verification: Java’s Vector Class, *Software Tools for Technology Transfer*, **3**(3), 2001, 332–352.
- [27] Jacobs, B.: A Formalisation of Java’s Exception Mechanism, *Proceedings of ESOP 2001* (D. Sands, Ed.), 2028, Springer-Verlag, 2001.
- [28] Jacobs, B., van den Berg, J., Huisman, M., van Barkum, M., Hensel, U., Tews, H.: Reasoning about Classes in Java (Preliminary Report), *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) ’98 (Vancouver, Canada)*, ACM, 1998, In *SIGPLAN Notices* 30(10).
- [29] Jacobs, B., Kiniry, J., Warnier, M.: Java Program Verification Challenges, in: Bonsangue et al. [20], 202–219.
- [30] Jacobs, B., Poll, E.: A Logic for the Java Modelling Language JML, *Fundamental Approaches to Software Engineering* (H. Hussmann, Ed.), 2029, Springer-Verlag, 2001.
- [31] Leavens, G. T., Leino, K. R. M., Müller, P.: Specification and Verification Challenges for Sequential Object-Oriented Programs, *Formal Aspects of Computing*, 2007, To appear.
- [32] Levin, G., Gries, D.: A Proof Technique for Communicating Sequential Processes, *Acta Informatica*, **15**(3), 1981, 281–302.
- [33] The LOOP project: Formal methods for object-oriented systems, <http://www.cs.kun.nl/~bart/LOOP/>, 2001.
- [34] Najm, E., Nestmann, U., Stevens, P., Eds.: *Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS ’03), Paris*, vol. 2884 of *Lecture Notes in Computer Science*, Springer-Verlag, November 2003.
- [35] Nipkow, T.: Hoare Logics in Isabelle/HOL, *Proof and System-Reliability* (H. Schwichtenberg, R. Steinbrüggen, Eds.), Kluwer, 2002.
- [36] Nipkow, T., von Oheimb, D.: Java-light is Type-Safe — Definitely, *Proceedings of POPL ’98*, ACM, 1998.
- [37] Nipkow, T., von Oheimb, D., Pusch, C.: μ Java: Embedding a Programming Language in a Theorem Prover, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999* (F. L. Bauer, R. Steinbrüggen, Eds.), IOS Press, 2000.
- [38] von Oheimb, D.: Axiomatic Semantics for Java^{light}, in: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications European Conference on Object-Oriented Programming (OOPSLA) (ECOOP)*, 2000.
- [39] von Oheimb, D.: *Axiomatic Semantics for Java^{light} in Isabelle/HOL*, Technical Report CSE 00-008, Oregon Graduate Institute, 2000.

- [40] von Oheimb, D.: Hoare Logic for Java in Isabelle/HOL, *Concurrency and Computation: Practice and Experience*, **13**(13), 2001, 1173–1214.
- [41] von Oheimb, D., Nipkow, T.: Machine-Checking the Java Specification: Proving Type-Safety, in: Alves-Foss [9].
- [42] von Oheimb, D., Nipkow, T.: Hoare Logic for NanoJava: Auxiliary Variables, Side Effects and Virtual Methods revisited, *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)* (L.-H. Eriksson, P. A. Lindsay, Eds.), 2391, Springer-Verlag, 2002.
- [43] Owicki, S., Gries, D.: An Axiomatic Proof Technique for Parallel Programs, *Acta Informatica*, **6**(4), 1976, 319–340.
- [44] Owre, S., Rushby, J. M., Shankar, N.: PVS: A Prototype Verification System, *Automated Deduction (CADE-11)* (D. Kapur, Ed.), 607, Springer-Verlag, 1992.
- [45] Paulson, L. C.: *The Isabelle Reference Manual*, Technical Report 283, University of Cambridge, Computer Laboratory, 1993.
- [46] Pierik, C.: *Validation Techniques for Object-Oriented Proof Outlines*, Ph.D. Thesis, Universiteit Utrecht, May 2006.
- [47] Pierik, C., de Boer, F. S.: A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts, in: Najm et al. [34], 64–78, An extended version appeared as University of Utrecht Technical Report UU-CS-2003-010.
- [48] Poetzsch-Heffter, A.: A Logic for the Verification of Object-Oriented Programs, *Proceedings of Programming Languages and Fundamentals of Programming* (R. Berghammer, F. Simon, Eds.), Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, November 1997, Bericht Nr. 9717.
- [49] Poetzsch-Heffter, A.: *Specification and Verification of Object-Oriented Programs*, Technische Universität München, January 1997, Habilitationsschrift.
- [50] Poetzsch-Heffter, A., Müller, P.: Logical Foundations for Typed Object-Oriented Languages, *Proceedings of PROCOMET '98* (D. Gries, W.-P. de Roever, Eds.), International Federation for Information Processing (IFIP), Chapman & Hall, 1998.
- [51] Poetzsch-Heffter, A., Müller, P.: A Programming Logic for Sequential Java, *Programming Languages and Systems* (S. Swierstra, Ed.), 1576, Springer, 1999.
- [52] Poll, E., van den Berg, J., Jacobs, B.: Specification of the JavaCard API in JML, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)* (J. Domingo-Ferrer, D. Chan, A. Watson, Eds.), Kluwer Acad. Publ., 2000.
- [53] Poll, E., van den Berg, J., Jacobs, B.: Formal specification of the Java Card API in JML: the APDU class, *Computer Networks*, **36**(4), 2001, 407–421.
- [54] Stärk, R., Schmid, J., Börger, E.: *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag, 2001.
- [55] Steffen, M.: *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*, Habilitation thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, 2006, Submitted 4th. July, accepted 7. February 2007.
- [56] Tucker, J. V., Zucker, J. I.: *Program Correctness over Abstract Data Types, with Error-State Semantics*, vol. 6 of *CWI Monograph Series*, North-Holland, 1988.
- [57] Warmer, J. B., Kleppe, A. G.: *The Object Constraint Language: Precise Modeling With UML*, Object Technology Series, Addison-Wesley, 1999.

A. Proofs of properties of substitutions and projection

Proof of Lemma 2.1 on page 402: By induction on the structure of local expressions and assertions. The base cases for local expressions are listed below, where the ones for instance and local variables are covered by the respective provisos of the lemma.

$$\begin{aligned}
\llbracket x[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} &= \llbracket z.x \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \sigma(\llbracket z \rrbracket_{\mathcal{G}}^{\omega, \sigma})(x) = \sigma(\omega(z))(x) = \llbracket x \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau} \\
\llbracket u[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} &= \llbracket u \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \omega(u) = \tau(u) = \llbracket u \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau} \\
\llbracket \text{this}[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} &= \llbracket z \rrbracket_{\mathcal{G}}^{\omega, \sigma} = [-2pt]\omega(z) = \llbracket \text{this} \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau} \\
\llbracket \text{null}[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} &= \text{null} = \llbracket \text{null} \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau} \\
\llbracket z'[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} &= \llbracket z' \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \omega(z') = \llbracket z' \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau} .
\end{aligned}$$

Compound expressions are treated by straightforward induction:

$$\begin{aligned}
&\llbracket f(e_1, \dots, e_n)[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} \\
&= f(\llbracket e_1[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma}, \dots, \llbracket e_n[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma}) \quad \text{semantics of assertions} \\
&= f(\llbracket e_1 \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau}, \dots, \llbracket e_n \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau}) \quad \text{by induction} \\
&= \llbracket f(e_1, \dots, e_n) \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau} \quad \text{semantics of assertions} .
\end{aligned}$$

For local assertions, negation and conjunction are straightforward. Unrestricted quantification $\exists z'. p$ in the local assertion language is only allowed for variables of type $t \in \{\text{Int}, \text{Bool}\}$ and for types composed from them, for which $\text{Val}_{\text{null}}^t(\sigma) = \text{Val}^t$. We get

$$\begin{aligned}
&\llbracket (\exists z'. p)[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{true} \\
&\iff \llbracket \exists z'. p[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{true} \quad \text{def. substitution} \\
&\iff \llbracket p[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega[z' \mapsto v], \sigma} = \text{true} \text{ for some } v \in \text{Val}^t \quad \text{assertion semantics} \\
&\iff \llbracket p \rrbracket_{\mathcal{L}}^{\omega[z' \mapsto v], \sigma(\omega(z)), \tau} = \text{true} \text{ for some } v \in \text{Val}^t \quad \text{by induction} \\
&\iff \llbracket \exists z'. p \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau} = \text{true} \quad \text{assertion semantics.}
\end{aligned}$$

For restricted quantification over elements of a sequence let $z' \in \text{LVar}^t$. Then

$$\begin{aligned}
&\llbracket (\exists z' \in e. p)[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{true} \\
&\iff \llbracket \exists z'. z' \in e[z/\text{this}] \wedge p[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} = \text{true} \quad \text{by definition} \\
&\iff \llbracket z' \in e[z/\text{this}] \wedge p[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega', \sigma} = \text{true} \quad \text{semantics} \\
&\quad \text{for some } v \in \text{Val}_{\text{null}}^t(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] \\
&\iff \left(\llbracket z' \rrbracket_{\mathcal{G}}^{\omega', \sigma} \in \llbracket e[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega', \sigma} \wedge \llbracket p[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega', \sigma} \right) = \text{true} \quad \text{semantics} \\
&\quad \text{for some } v \in \text{Val}_{\text{null}}^t(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] \\
&\iff \left(\llbracket z' \rrbracket_{\mathcal{L}}^{\omega', \sigma(\omega(z)), \tau} \in \llbracket e \rrbracket_{\mathcal{L}}^{\omega', \sigma(\omega(z)), \tau} \wedge \llbracket p \rrbracket_{\mathcal{L}}^{\omega', \sigma(\omega(z)), \tau} \right) = \text{true} \quad \text{by induction} \\
&\quad \text{for some } v \in \text{Val}_{\text{null}}^t(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] \\
&\iff \llbracket (z' \in e) \wedge p \rrbracket_{\mathcal{L}}^{\omega', \sigma(\omega(z)), \tau} = \text{true} \quad \text{semantics} \\
&\quad \text{for some } v \in \text{Val}_{\text{null}}^t(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] \\
&\iff \llbracket \exists z' \in e. p \rrbracket_{\mathcal{L}}^{\omega, \sigma(\omega(z)), \tau} = \text{true} \quad \text{semantics} .
\end{aligned}$$

The last equation uses the assumption that the local state τ and the instance state $\sigma(\omega(z))$ assign values from $Val_{null}(\sigma)$ to all variables, i.e., e does not refer to values of non-existing objects (see Lemma A.1). Consequently, $v \in Val_{null}^t$ together with $\llbracket z' \in e \rrbracket_{\mathcal{L}}^{\omega[z' \mapsto v], \sigma(\omega(z)), \tau} = true$ implies $v \in Val_{null}^t(\sigma)$. The case for restricted quantification over subsequences is analogous. \square

Proof of Lemma 6.1 on page 424: We proceed by straightforward induction on the structure of local assertions. Let $\acute{\sigma}_{inst} = \acute{\sigma}_{inst}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{L}}^{\omega, \acute{\sigma}_{inst}, \acute{\tau}}]$ and $\acute{\tau} = \acute{\tau}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{L}}^{\omega, \acute{\sigma}_{inst}, \acute{\tau}}]$. In the case for local variables $u = y_i$ we get

$$\begin{aligned} \llbracket u[\vec{e}/\vec{y}] \rrbracket_{\mathcal{L}}^{\omega, \acute{\sigma}_{inst}, \acute{\tau}} &= \llbracket e_i \rrbracket_{\mathcal{L}}^{\omega, \acute{\sigma}_{inst}, \acute{\tau}} \\ &= \acute{\tau}(u) \\ &= \llbracket u \rrbracket_{\mathcal{L}}^{\omega, \acute{\sigma}_{inst}, \acute{\tau}}. \end{aligned}$$

For instance variables $x = y_i$ similarly:

$$\begin{aligned} \llbracket x[\vec{e}/\vec{y}] \rrbracket_{\mathcal{L}}^{\omega, \acute{\sigma}_{inst}, \acute{\tau}} &= \llbracket e_i \rrbracket_{\mathcal{L}}^{\omega, \acute{\sigma}_{inst}, \acute{\tau}} \\ &= \acute{\sigma}_{inst}(x) \\ &= \llbracket x \rrbracket_{\mathcal{L}}^{\omega, \acute{\sigma}_{inst}, \acute{\tau}}. \end{aligned}$$

The remaining cases are straightforward. \square

Proof of Lemma 6.2 on page 425: Let $\acute{\omega} = \acute{\omega}[\vec{y} \mapsto \llbracket \vec{E} \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}}]$ and $\acute{\sigma} = \acute{\sigma}[\llbracket z \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} . \vec{y} \mapsto \llbracket \vec{E} \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}}]$. We proceed by induction on the structure of global expressions and assertions. The base cases for null and z' are straightforward. For the induction cases, we start with the crucial one for qualified reference to instance variables. For expressions $E'.x[\vec{E}/z.\vec{y}]$ with x not in \vec{y} the property holds by induction. So assume that x is in \vec{y} :

$$\llbracket (E'.y_i)[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} = \llbracket \text{if } E'[\vec{E}/z.\vec{y}] = z \text{ then } E_i \text{ else } (E'[\vec{E}/z.\vec{y}]).y_i \text{ fi} \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}}.$$

This conditional assertion evaluates to $\llbracket E_i \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}}$ if $\llbracket E'[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} = \llbracket z \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}}$ and to $\llbracket (E'[\vec{E}/z.\vec{y}]).y_i \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}}$ otherwise. So in the first case we get

$$\begin{aligned} \llbracket (E'.y_i)[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} &= \llbracket E_i \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} \\ &= \acute{\sigma}(\llbracket z \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}})(y_i) && \text{by def. of } \acute{\sigma} \\ &= \acute{\sigma}(\llbracket E'[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}})(y_i) && \text{by the case assumption} \\ &= \acute{\sigma}(\llbracket E' \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}})(y_i) && \text{by induction} \\ &= \llbracket E'.y_i \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} && \text{by def. of } \llbracket - \rrbracket_{\mathcal{G}}. \end{aligned}$$

If otherwise $\llbracket E'[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} \neq \llbracket z \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}}$, then

$$\begin{aligned} \llbracket (E'.y_i)[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} &= \llbracket (E'[\vec{E}/z.\vec{y}]).y_i \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} \\ &= \acute{\sigma}(\llbracket E'[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}})(y_i) && \text{by def. of } \llbracket - \rrbracket_{\mathcal{G}} \\ &= \acute{\sigma}(\llbracket E'[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}})(y_i) && \text{case assumption+def. } \acute{\sigma} \\ &= \acute{\sigma}(\llbracket E' \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}})(y_i) && \text{by induction} \\ &= \llbracket E'.y_i \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} && \text{by def. of } \llbracket - \rrbracket_{\mathcal{G}}. \end{aligned}$$

For operator expressions we get:

$$\begin{aligned}
& \llbracket (f(E_1, \dots, E_n))[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} \\
&= \llbracket f(E_1[\vec{E}/z.\vec{y}], \dots, E_n[\vec{E}/z.\vec{y}]) \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} && \text{def. substitution} \\
&= f(\llbracket E_1[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}}, \dots, \llbracket E_n[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}}) && \text{def. } \llbracket - \rrbracket_{\mathcal{G}} \\
&= f(\llbracket E_1 \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}}, \dots, \llbracket E_n \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}}) && \text{by induction} \\
&= \llbracket f(E_1, \dots, E_n) \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} && \text{def. } \llbracket - \rrbracket_{\mathcal{G}} .
\end{aligned}$$

For global assertions, the cases of negation and conjunction are straightforward. For quantification,

$$\begin{aligned}
& \llbracket (\exists z'. P)[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} = true \\
\iff & \llbracket \exists z'. P[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} = true && \text{def. substitution} \\
\iff & \llbracket P[\vec{E}/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\dot{\omega}[z' \mapsto v], \dot{\sigma}} = true \text{ for some } v \in Val_{null}(\dot{\sigma}) && \text{def. } \llbracket - \rrbracket_{\mathcal{G}} \\
\iff & \llbracket P \rrbracket_{\mathcal{G}}^{\dot{\omega}[z' \mapsto v], \dot{\sigma}} = true \text{ for some } v \in Val_{null}(\dot{\sigma}) && \text{by induction} \\
\iff & \llbracket \exists z'. P \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} = true, && Val(\dot{\sigma}) = Val(\dot{\sigma})
\end{aligned}$$

where z' is not in \vec{y} (otherwise the substitution renames z'). \square

Lemma A.1. Let σ be a global state and ω a logical environment referring only to values existing in σ . Then $\llbracket E \rrbracket_{\mathcal{G}}^{\omega, \sigma} \in Val_{null}(\sigma)$ for all global expressions $E \in GExp$ that can be evaluated in the context of ω and σ .

Proof of Lemma A.1: By structural induction on the global assertion. The case for logical variables $z \in LVar^t$ is immediate by the assumption about ω , the ones for null and operator expressions are trivial, respectively follows by induction. For qualified references $E.x$ with $E \in GExp^c$ and x an instance variable of type t in class c , if $E.x$ can be evaluated in the context of ω and σ , then $\llbracket E \rrbracket_{\mathcal{G}}^{\omega, \sigma} \neq null$. Hence by induction $\llbracket E \rrbracket_{\mathcal{G}}^{\omega, \sigma} \in Val_{null}(\sigma)$, more specifically $\llbracket E \rrbracket_{\mathcal{G}}^{\omega, \sigma} \in Val(\sigma)$. Therefore by definition of global states $\sigma(\llbracket E \rrbracket_{\mathcal{G}}^{\omega, \sigma})(x) \in Val_{null}(\sigma)$. \square

Proof of Lemma 2.2 on page 409: We prove the lemma by structural induction on global assertions. Assume a global state $\dot{\sigma}$, and let $\dot{\sigma} = \dot{\sigma}[\alpha \mapsto \sigma_{inst}^{c, init}]$ be an extension of $\dot{\sigma}$ with a new object $\alpha \in Val^c$, $\alpha \notin Val(\dot{\sigma})$. Assume furthermore a logical environment ω referring only to values existing in $\dot{\sigma}$, and let v be the sequence consisting of all elements of $\bigcup_c Val_{null}^c(\dot{\sigma})$. Let finally P be a global assertion, $z' \in LVar^{list\ Object}$ a logical variable not occurring in P , and $\dot{\omega} = \dot{\omega}[z' \mapsto v]$. Since z' is fresh in P , we have for all logical variables z in P that $\llbracket z \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} = \dot{\omega}(z) = \dot{\omega}(z) = \llbracket z \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} = \llbracket z \downarrow z' \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}}$. For qualified references to instance variables, the argument is as follows:

$$\begin{aligned}
\llbracket E.x \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} &= \dot{\sigma}(\llbracket E \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}})(x) && \text{semantics} \\
&= \dot{\sigma}(\llbracket E \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}})(x) && \llbracket E \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} \neq \alpha \text{ by Lemma A.1 and } \alpha \notin Val(\dot{\sigma}) \\
&= \dot{\sigma}(\llbracket E \downarrow z' \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}})(x) && \text{by induction} \\
&= \llbracket (E \downarrow z').x \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} && \text{semantics} \\
&= \llbracket (E.x) \downarrow z' \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} && \text{def. } \downarrow z' .
\end{aligned}$$

The interesting case is the one for quantification. For $z \in LVar^t$:

$$\begin{array}{llll}
& \dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} \exists z. P & & \\
\iff & \dot{\omega}[z \mapsto u], \dot{\sigma} \models_{\mathcal{G}} P \text{ for some } u \in Val_{null}^t(\dot{\sigma}) & \text{semantics} & \\
\iff & \dot{\omega}[z \mapsto u], \dot{\sigma} \models_{\mathcal{G}} P \downarrow z' \text{ for some } u \in Val_{null}^t(\dot{\sigma}) & \text{induction} & \\
\iff & \dot{\omega}[z \mapsto u], \dot{\sigma} \models_{\mathcal{G}} \text{obj}(z) \subseteq z' \wedge P \downarrow z' & \text{obj}(u) \subseteq v & \\
& \text{for some } u \in Val_{null}^t(\dot{\sigma}) & & \\
\iff & \dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} \exists z. \text{obj}(z) \subseteq z' \wedge P \downarrow z' & \text{semantics} & \\
\iff & \dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} (\exists z. P) \downarrow z'. & &
\end{array}$$

The remaining cases are straightforward. \square

B. Soundness proof

This section contains the inductive proof of soundness of the proof method. We start with some ancillary lemmas about basic invariant properties of proof outlines, for instance properties of the built-in auxiliary variables added in the transformation. Afterwards, we show soundness of the proof system.

B.1. Invariant properties

Lemma B.1. Let $prog'$ be a proof outline for a program $prog$. Then $\langle T, \sigma \rangle$ is a reachable configuration of $prog$ iff there exists a reachable configuration $\langle T', \sigma' \rangle$ of $prog'$ with $\langle T' \downarrow prog, \sigma' \downarrow prog \rangle = \langle T, \sigma \rangle$.

Proof of the transformation Lemma B.1: We proceed for both directions by straightforward induction on the length of reduction. The only interesting property of the transformation is the representation of notification by a single auxiliary assignment of the notifier. For this case we use Lemma B.3 showing soundness of the representation of the wait and notified sets by the auxiliary instance variables wait and notified. \square

Lemma B.2. (Identification)

Let $\langle T, \sigma \rangle$ be a reachable configuration of a proof outline. Then

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

where $\text{proj}(v, i)$ is the i th component of the tuple v .

Proof of Lemma B.2 on the facing page: All parts by straightforward induction on the steps of proof outlines. \square

Lemma B.3. (Lock, Wait, Notify)

Let $\langle T, \sigma \rangle$ be a reachable configuration of a proof outline for the original program $prog$, $\alpha \in Val(\sigma)$ an object identity, and let $\xi = (\alpha_0, \tau_0, stm_0) \circ \xi^l \in T$. Let furthermore n be the number synchronized method executions of ξ in α , i.e., $n = |\{(\alpha, \tau, stm) \in \xi \mid stm \text{ synchr.}\}|$. Then

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

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

Proof of Lemma B.3: The cases 2a and 2b are satisfied by the definition of the projection operator. Inductivity for the cases 2c and 2d are easy to show using Lemma B.2 and the cases 2a and 2b of this lemma. If the order of the elements is unimportant, in the following we also use set notation for the values of the wait and notified variables. Correctness of the projection operation uses the results of this lemma and is formulated in Lemma B.1. For the other cases we proceed by induction on the length of the run $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \dot{T}, \dot{\sigma} \rangle$ of the proof outline $prog'$.

In the base case of an initial configuration $\langle T_0, \sigma_0 \rangle$ (see page 398), the set T_0 contains exactly one thread (α, τ, stm) , executing the non-synchronized main-statement of the program, i.e., $\neg owns(T_0 \downarrow prog, \alpha)$, and initially the lock of the only object α is set to $free$. Furthermore, the instance variables wait and notified of the initial object are set to \emptyset , and the $wait$ and $notified$ sets of the semantics are also empty.

For the inductive step, assume $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \dot{T}, \dot{\sigma} \rangle \longrightarrow \langle \dot{T}', \dot{\sigma}' \rangle$. We distinguish on the kind of the last computation step.

Case: $CALL_{start}$, $CALL_{start}^{skip}$, $RETURN_{run}$, TRY , $FINALLY$, YRT , $THROW_1$, $THROW_2$, $THROW_3$, $THROW_5$

In these cases none of the concerned variables or predicates are touched, and the property follows directly by induction.

Case: ASS_{inst} , ASS_{loc}

Note that this case handles assignments, but not the observations of communication, object creation, and exception handling. Remember furthermore that the signaling mechanism is implemented in proof outlines by auxiliary assignments, and thus this case covers also the rules $SIGNAL$, $SIGNAL_{skip}$, and $SIGNALALL$.

If the assignment is not in a notify- or in a notifyAll-method representing notification, then the case is analogous to the above one.

Assume first that the assignment in the last computation step represents notification in a notify-method of the proof outline. If the wait set $\delta(\alpha)(\text{wait})$ is empty, then no notification takes place; the property follows directly by induction. Thus assume that the wait set is not empty. I.e., a thread $\xi_1 \in \hat{T}$ notifies another thread $\xi_2 = (\alpha_2, \tau, stm) \circ \xi'_2 \in \hat{T}$ in the *wait* set of α . Remember that notification is represented by a single assignment of the notifier, and thus the stack of the notified thread ξ_2 does not change. However, according to the projection definition, as the notifier changes the value of *wait* of α , the projection $\xi_2 \downarrow \text{prog}$ represents a thread being in the wait set in $\langle \hat{T}, \delta \rangle$ and being in the notified set in $\langle \hat{T}, \delta \rangle$.

The only relevant effect of the step is moving $(\alpha_2, n) \in \delta(\alpha)(\text{wait})$ from the wait set into the notified set of α , where n is by induction the number of synchronized invocations of ξ_2 in α . Thus the properties 1a, 1b and 2e are automatically invariant. Induction implies also uniqueness of the representation of the wait and notified sets, i.e., α_2 is contained neither in $\delta(\alpha)(\text{notified})$ nor in $\delta(\alpha)(\text{wait})$. Thus moving the thread of α_2 from the wait into the notified set does not violate uniqueness of the representation.

The case for the assignment in the notifyAll-method is analogous, with the difference that all threads in the wait set get notified by ξ_1 . The notifier sets the value of the auxiliary instance variable *notified* of α to $\delta(\alpha)(\text{notified}) \cup \delta(\alpha)(\text{wait})$, whereas the corresponding *wait* variable gets the value \emptyset . By induction we have $\delta(\alpha)(\text{notified}) \cap \delta(\alpha)(\text{wait}) = \emptyset$, and thus the required properties are invariant under notification.

Case: NEW

Assume that the last step creates a new object, and executes the corresponding observation. Let $\alpha \in \text{dom}(\delta)$. Then α either references the newly created object, or $\alpha \in \text{dom}(\delta)$. In the first case $\alpha \notin \text{dom}(\delta)$, and by the definition of global configurations (see page 397) there is no local configuration $(\alpha, \tau, stm) \in \hat{T}$, and the *wait* and *notified* set of α in \hat{T} are empty. Since the last step doesn't add any local configurations to \hat{T} , we have $\alpha \neq \beta$ for all $(\beta, \tau, stm) \in \hat{T}$ and thus $\neg \text{owns}(\hat{T} \downarrow \text{prog}, \alpha)$. Since the lock of the new object is initialized to *free*, and *wait* and *notified* of α get the value \emptyset , the required property holds for the new object. In the second case, if $\alpha \in \text{dom}(\delta)$, the property follows directly by induction.

Case: CALL

Let $\alpha \in \text{dom}(\delta)$. Then also $\alpha \in \text{dom}(\delta)$. If α is not the callee object, then the property holds directly by induction. If α is the callee object, the only new local configuration (α, τ, stm) in \hat{T} represents the execution of the invoked method.

If the invoked method is non-synchronized, then the property follows by induction (invocations of monitor methods are covered by the $\text{CALL}_{\text{monitor}}$ case below). In the case of a synchronized method, let $\xi \in \hat{T}$ be the executing thread. The antecedent $\neg \text{owns}(\hat{T} \setminus \{\xi\} \downarrow \text{prog}, \alpha)$ implies by induction that, if there is no local configuration in the thread's stack executing a synchronized method of α then $\delta(\alpha)(\text{lock}) = \text{free}$, and $\delta(\alpha)(\text{lock}) = (\alpha_0, n)$ otherwise, where $(\alpha_0, \tau_0, stm_0)$ is the deepest configuration in the thread's stack and n the number of synchronized method invocations in the stack ξ . If in the state prior to the method invocation $\delta(\alpha)(\text{lock}) = \text{free}$, then (α, τ, stm) is the only local configuration in \hat{T} representing the execution of a synchronized method of α by a thread not in the *wait* or *notified* sets of α . Furthermore, the callee observation sets $\delta(\alpha)(\text{lock}) = (\alpha_0, 1)$, and thus the required property holds. In the second case, using the fact that the callee configuration is on top of its stack, the callee observation

changes $\delta(\alpha)(\text{lock}) = (\alpha_0, n)$ to $\sigma(\alpha)(\text{lock}) = (\alpha_0, n + 1)$, and we get the property by Lemma B.2 and by induction.

Case: CALL_{monitor}

Similarly to the case CALL, for $\alpha \in \text{dom}(\sigma)$ also $\alpha \in \text{dom}(\delta)$, and if α is not the callee object, then the property holds by induction. In the case of the non-synchronized notify- and notifyAll-methods, none of the concerned variables or predicates are touched, and thus the property holds by induction again. So let $\xi \in \dot{T}$ be the executing thread invoking the non-synchronized wait-method of α .

The antecedent $\text{owns}(\xi \downarrow \text{prog}, \alpha)$ implies by induction $\delta(\alpha)(\text{lock}) = (\alpha_0, n)$, where $(\alpha_0, \tau_0, \text{stm}_0)$ is the deepest configuration in the stack ξ and n is the number of its synchronized method invocations in α . Furthermore, since ξ does not yet execute a wait-method prior to the call, from $\xi \notin \text{wait}(\dot{T} \downarrow \text{prog}, \alpha) \cup \text{notified}(\dot{T} \downarrow \text{prog}, \alpha)$ we conclude by induction that α_0 is contained neither in wait nor in notified of α in δ .

The execution places the thread into α 's wait set and, since at most one thread can own a lock at a time, it gives the lock of α free, i.e., we have $\neg \text{owns}(\dot{T} \downarrow \text{prog}, \alpha)$. The corresponding callee observation extends $\delta(\alpha)(\text{wait})$ with (α_0, n) , and sets the lock-value of α to *free*. Thus the case follows by induction.

Case: RETURN

Assume $\alpha \in \text{dom}(\sigma) = \text{dom}(\delta)$. If α is not the callee object, or if the invoked method is non-synchronized, then the property holds directly by induction. Note that returning from the wait-method is covered by the RETURN_{wait} case below. So let $\xi \in \dot{T}$ be the thread of α_0 returning from a synchronized method of α ; we denote the thread after execution by $\xi' \in \dot{T}$.

Since ξ is neither in the wait nor in the notified set of α , we get by definition $\text{owns}(\xi \downarrow \text{prog}, \alpha)$ prior to execution. If the given method is the only synchronized method of α executed by ξ , then in the successor configuration $\neg \text{owns}(\xi' \downarrow \text{prog}, \alpha)$, and from the invariant property that at most one thread can own a lock at a time we imply $\neg \text{owns}(\dot{T} \downarrow \text{prog}, \alpha)$. Otherwise, if ξ has reentrant synchronized method invocations in α , then the thread doesn't give the lock free upon return, i.e., in the successor state we still have $\text{owns}(\xi' \downarrow \text{prog}, \alpha)$.

Using $\text{owns}(\xi \downarrow \text{prog}, \alpha)$, we get by induction $\delta(\alpha)(\text{lock}) = (\alpha_0, n)$, where n is the number of invocations of synchronized methods of α by ξ . The auxiliary variable lock of α is set by the callee augmentation to *free*, if $n = 1$, and to $(\alpha_0, n - 1)$, otherwise. Since the auxiliary variables wait and notified are not touched, the property follows by induction.

Case: RETURN_{wait}

Assume that the thread $\xi \in \dot{T}$ of an object α_0 is returning from the wait-method of $\alpha \in \text{dom}(\sigma) = \text{dom}(\delta)$; we denote the thread after execution by $\xi' \in \dot{T}$.

The semantics assures $\neg \text{owns}(\dot{T} \downarrow \text{prog}, \alpha)$ and by definition $\xi \in \text{notified}(\dot{T} \downarrow \text{prog}, \alpha)$. We get by induction $\delta(\alpha)(\text{lock}) = \text{free}$ and $(\alpha_0, n) \in \delta(\alpha)(\text{notified})$, where n is the number of invocations of synchronized methods of α by ξ . After returning, the thread gets removed from the notified-set of α and gathers the lock of α , i.e., $\xi' \notin \text{notified}(\dot{T} \downarrow \text{prog}, \alpha)$ and $\text{owns}(\xi' \downarrow \text{prog}, \alpha)$.

The augmentation of the wait-method removes (α_0, n) from $\delta(\alpha)(\text{notified})$; from the uniqueness of the representation follows $\alpha_0 \neq \beta$ for all $(\beta, m) \in \delta(\alpha)(\text{notified})$. Furthermore, the observation sets the lock of α to (α_0, n) , by which we get the required property.

Case: THROW₄

This case is analogous to the case RETURN. Remember that the observations of throw statements outside

try-catch-finally blocks in synchronized methods decrement the lock value. \square

Lemma B.4. (Started)

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

Proof of Lemma B.4: Straightforward by the definition of augmentation. \square

B.2. Proof of the soundness theorem

Proof of the soundness Theorem 7.1 on page 429: We prove the theorem by induction on the length of the computation, simultaneously for all parts of Definition 7.1.

For the initial case assume $dom(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma_{inst}^{init}[\text{this} \mapsto \alpha]$, $\tau_0 = \tau^{init}[\text{thread} \mapsto \alpha]$, and let $\{p_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{p_3\} stm$ be the main statement. Then the initial configuration $\langle T'_0, \sigma'_0 \rangle$ of the proof outline satisfies the following: $\sigma'_0 = \sigma_0[\alpha.\vec{y}_2 \mapsto \llbracket \vec{e}_2 \rrbracket_{\mathcal{E}}^{\sigma_0(\alpha), \tau_0}]$, and for the stack we have $T'_0 = \{(\alpha, \tau'_0, stm)\}$ with $\tau'_0 = \tau_0[\vec{y}_2 \mapsto \llbracket \vec{e}_2 \rrbracket_{\mathcal{E}}^{\sigma_0(\alpha), \tau_0}]$.

Let ω be a logical environment referring only to values existing in σ_0 . As in σ_0 there exists exactly one object α being in its initial instance state, we have

$$\omega[z \mapsto \alpha], \sigma_0 \models_G \text{InitState}(z) \wedge \forall z'. z' = \text{null} \vee z = z',$$

where z is of the type of the main class, and z' is a logical variable of type Object. Using the initial correctness condition we get

$$\omega[z \mapsto \alpha], \sigma_0 \models_G (GI \wedge P_3(z) \wedge I(z)) \circ f_{obs} \circ f_{init}$$

with I the class invariant of α , \vec{v} the local variables of the run-method of the main class, and

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

Applying Lemma 6.2, we get for the global invariant $\omega', \sigma'_0 \models_G GI$ for $\omega' = \omega[z \mapsto \alpha][\vec{v} \mapsto \tau'_0(\vec{v})]$. Since GI may not contain free logical variables, its value does not depend on the logical environment, and therefore $\omega, \sigma'_0 \models_G GI$.

Similarly for the local property p_3 , we get with Lemma 6.2 that $\omega', \sigma'_0 \models_{\mathcal{L}} P_3(z)$. With Lemma 2.1 we get $\omega', \sigma'_0(\alpha), \tau'_0 \models_{\mathcal{L}} pre(stm)$. Since $pre(stm)$ does not contain free logical variables, we get finally $\omega, \sigma'_0(\alpha), \tau'_0 \models_{\mathcal{L}} pre(stm)$. Part 3 is analogous.

For the inductive step, assume $\langle T_0, \sigma_0 \rangle \xrightarrow{*} \langle \dot{T}, \dot{\sigma} \rangle \longrightarrow \langle \dot{T}', \dot{\sigma}' \rangle$ such that $\langle \dot{T}, \dot{\sigma} \rangle$ satisfies the conditions of Definition 7.1. Let ω be a logical environment referring only to values existing in $\dot{\sigma}$. We distinguish on the kind of the computation step $\langle \dot{T}, \dot{\sigma} \rangle \longrightarrow \langle \dot{T}', \dot{\sigma}' \rangle$.

If the computation step is executed by a single local configuration, we use the local correctness conditions for inductivity of the executing local configuration's properties, and the interference freedom test for all other local configurations and the class invariants in $\langle \dot{T}', \dot{\sigma}' \rangle$. For communication, invariance

for the executing partners and the global invariant is shown using the cooperation test for communication. Communication itself does not affect the global state; invariance of the remaining properties under the corresponding observations is shown again with the help of the interference freedom test. The case for throwing exceptions outside try-blocks is similar. Finally for object creation, invariance for the global invariant, the creator local configuration, the created object's class invariant is assured by the conditions of the cooperation test for object creation; all other properties are shown to be invariant using the interference freedom test.

Case: ASS_{inst}, ASS_{loc}

Note that signaling is represented in proof outlines by auxiliary assignments, thus this case covers also the rules SIGNAL, SIGNALALL, and SIGNAL_{skip}. Note furthermore that this case does not cover observations of communication, object creation, or exception throwing and handling.

Let the last computation step be the execution of an assignment in the local configuration $(\alpha, \hat{\tau}_1, \vec{y} := \vec{e}; stm_1) \in \dot{T}$ resulting in $(\alpha, \hat{\tau}_1, stm_1) \in \dot{T}$. According to the semantics, $\hat{\tau}_1 = \hat{\tau}_1[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1}]$ and $\hat{\sigma} = \hat{\sigma}[\alpha.\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1}]$.

Since assignments, that does not observe object creation, communication, or exception throwing, don't change the values of variables occurring in *GI*, part (2) is satisfied.

For part (1), assume $(\beta, \tau_2, stm_2) \in \dot{T}$. If $(\beta, \tau_2, stm_2) = (\alpha, \hat{\tau}_1, stm_1)$ is the executing local configuration, then by induction $\omega, \hat{\sigma}(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$. The local correctness condition implies that $\omega, \hat{\sigma}(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} pre(stm_1)[\vec{e}/\vec{y}]$. Using the properties of the local substitution formulated in Lemma 6.1 we get $\omega, \hat{\sigma}(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} pre(stm_1)$.

If otherwise (β, τ_2, stm_2) is not the executing local configuration, then it is contained in \dot{T} . If $\alpha \neq \beta$, i.e., the execution didn't take place in β , then $\hat{\sigma}(\beta) = \hat{\sigma}(\beta)$, and thus $\omega, \hat{\sigma}(\beta), \tau_2 \models_{\mathcal{L}} pre(stm_2)$ by induction. Otherwise let τ be $\hat{\tau}_1[\vec{v}' \mapsto \tau_2(\vec{v})]$, where $\vec{v} = dom(\tau_2)$ and \vec{v}' fresh. Then Lemma B.2, the induction assumptions, and the definition of interleavable imply

$$\omega, \hat{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e}) \wedge pre'(stm_2) \wedge interleavable(pre(stm_2), \vec{y} := \vec{e}),$$

and with the interference freedom test we get $\omega, \hat{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_2)[\vec{e}/\vec{y}]$. Using the substitution Lemma 6.1 and the fact that, due to the renaming mechanism, no variables in \vec{v}' may occur in \vec{y} , yields $\omega, \hat{\sigma}(\alpha), \tau_2 \models_{\mathcal{L}} pre(stm_2)$.

Part (3) is similar, using the fact that the class invariant may contain instance variables only, and thus its evaluation doesn't depend on the local state.

Case: CALL

Let $(\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1) \in \dot{T}$ be the caller configuration prior to method invocation, and let $(\alpha, \hat{\tau}_1, stm'_1) \in \dot{T}$ and $(\beta, \hat{\tau}_2, stm_2) \in \dot{T}$ be the local configurations of the caller and the callee after execution. Let furthermore $\langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} stm_2$ be the invoked method's body and \vec{u} its formal parameters. Directly after communication the callee has the local state $\hat{\tau}_2 = \tau^{init}[\vec{u} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1}]$; after the caller observation, the global state is $\hat{\sigma} = \hat{\sigma}[\alpha.\vec{y}_1 \mapsto \llbracket \vec{e}_1 \rrbracket_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1}]$ and the caller's local state is updated to $\hat{\tau}_1 = \hat{\tau}_1[\vec{y}_1 \mapsto \llbracket \vec{e}_1 \rrbracket_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1}]$. Finally, the callee observation updates its local state to $\hat{\tau}_2 = \hat{\tau}_2[\vec{y}_2 \mapsto \llbracket \vec{e}_2 \rrbracket_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$ and the global state to $\hat{\sigma} = \hat{\sigma}[\beta.\vec{y}_2 \mapsto \llbracket \vec{e}_2 \rrbracket_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$. Let \vec{v}_1 denote $dom(\hat{\tau}_1)$ and assume $\hat{\omega} = \omega[z \mapsto \alpha][z' \mapsto \beta][\vec{v}_1 \mapsto \hat{\tau}_1(\vec{v}_1)]$.

The semantics assures $\alpha \neq null$ and $\beta = \llbracket e_0 \rrbracket_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1} \neq null$, and we get with Lemma 2.1 and the definition of $\hat{\omega}$ that $\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} z \neq null \wedge z' \neq null \wedge E_0(z) = z'$.

If the method is synchronized and ξ is the stack of the executing thread in \dot{T} , then according to the transition rule $\neg\text{owns}(\dot{T} \setminus \{\xi\} \downarrow \text{prog}, \beta)$. Using Lemma B.3 and Lemma B.2 we get $\dot{\sigma}(\beta)(\text{lock}) = \text{free} \vee \text{thread}(\dot{\sigma}(\beta)(\text{lock})) = \dot{\tau}_1(\text{thread})$ and thus $\dot{\omega}, \dot{\sigma} \models_G z'.\text{lock} = \text{free} \vee \text{thread}(z'.\text{lock}) = \text{thread}$.

In the following let $p_1 = \text{pre}(u_{\text{ret}} := e_0.m(\vec{e}))$, $p_2 = \text{pre}(\vec{y}_1 := \vec{e}_1)$, $p_3 = \text{post}(\vec{y}_1 := \vec{e}_1)$, $q_1 = I_q$, $q_2 = \text{pre}(\vec{y}_2 := \vec{e}_2)$, and $q_3 = \text{post}(\vec{y}_2 := \vec{e}_2)$, where I_q is the class invariant of the callee. Then we have by induction $\dot{\omega}, \dot{\sigma} \models_G GI$, for the class invariant $\dot{\omega}, \dot{\sigma}(\beta), \dot{\tau}_1 \models_{\mathcal{L}} I_q$, and for the precondition of the call $\dot{\omega}, \dot{\sigma}(\alpha), \dot{\tau}_1 \models_{\mathcal{L}} p_1$. Using the lifting lemma, the cooperation test for communication implies

$$\begin{aligned} \dot{\omega}, \dot{\sigma} \models_G & (P_2(z) \wedge Q'_2(z'))[\vec{E}(z), \text{Init}(\vec{v})/\vec{u}', \vec{v}'] \wedge \\ & (GI \wedge P_3(z) \wedge Q'_3(z'))[E'_2(z')/z'.\vec{y}'_2][E_1(z)/z.\vec{y}_1][\vec{E}(z), \text{Init}(\vec{v})/\vec{u}', \vec{v}'], \end{aligned}$$

where \vec{v} contains the local variables of the callee without the formal parameters \vec{u} . Using the lifting lemma again but in the reverse direction and Lemma 6.2 results $\omega, \acute{\sigma} \models_G GI$, and thus part (2). Note that in the annotation no free logical variables occur, and thus the values of assertions in a proof outline do not depend on the logical environment. Furthermore, using the same lemmas we get

$$\begin{aligned} \omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_2 & \quad \omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} q_2 \\ \omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3 & \quad \omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} q_3. \end{aligned}$$

Thus part (1) is satisfied for the local configurations involved in the last computation step. All other configurations $(\gamma, \tau_3, \text{stm}_3)$ in \dot{T} are also in \dot{T} . If $\gamma \neq \alpha$ and $\gamma \neq \beta$, then $\dot{\sigma}(\gamma) = \acute{\sigma}(\gamma)$, and thus $\omega, \acute{\sigma}(\gamma), \tau_3 \models_{\mathcal{L}} \text{pre}(\text{stm}_3)$ by induction.

Assume next $\gamma = \alpha$ and $\alpha \neq \beta$, and let τ be $\dot{\tau}_1[\vec{v}' \mapsto \tau_3(\vec{v})]$, where $\vec{v} = \text{dom}(\tau_3)$. Then Lemma B.2, the induction assumptions, and the definition of the assertion interleavable imply with the interference freedom test $\omega, \acute{\sigma}(\alpha), \tau \models_{\mathcal{L}} \text{pre}'(\text{stm}_3)[\vec{e}_1/\vec{y}_1]$. The substitution Lemma 6.1 and the fact that, due to the renaming mechanism, no local variables in \vec{v}' occur in \vec{y}_1 , yield $\omega, \acute{\sigma}(\alpha), \tau_3 \models_{\mathcal{L}} \text{pre}(\text{stm}_3)$. Now, since $\beta \neq \alpha$, the callee observation also does not change the caller's instance state, and we have $\dot{\sigma}(\alpha) = \acute{\sigma}(\alpha)$. Thus we get $\omega, \acute{\sigma}(\alpha), \tau_3 \models_{\mathcal{L}} \text{pre}(\text{stm}_3)$.

The case $\gamma = \beta$ and $\alpha \neq \beta$ is similar. Communication and caller observation do not change the instance state of β , i.e., $\dot{\sigma}(\beta) = \acute{\sigma}(\beta)$. The interference freedom test results $\omega, \acute{\sigma}(\beta), \tau \models_{\mathcal{L}} \text{pre}'(\text{stm}_3)[\vec{e}_2/\vec{y}_2]$ with $\tau = \dot{\tau}_2[\vec{v}' \mapsto \tau_3(\vec{v})]$. Due to the renaming mechanism, we conclude with the local substitution lemma that $\omega, \acute{\sigma}(\beta), \acute{\tau} \models_{\mathcal{L}} \text{pre}'(\text{stm}_3)$ with $\acute{\tau}(\vec{v}') = \tau_3(\vec{v})$, and thus $\omega, \acute{\sigma}(\beta), \tau_3 \models_{\mathcal{L}} \text{pre}(\text{stm}_3)$.

For the last case $\gamma = \alpha = \beta$ note that, according to the restrictions on the augmentation, the caller may not change the instance state. Thus the same arguments as for $\gamma = \beta$ and $\alpha \neq \beta$ apply. I.e., part (1) is satisfied.

Part (3) is analogous: The interference freedom test implies $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} I_p$, where I_p is the class invariant of the caller. Since I_p may contain instance variables only, its evaluation doesn't depend on the local state. Similarly for the callee, $\omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} I_q$. The state of other objects is not changed in the last computation step, and we get the required property.

Case: $\text{CALL}_{\text{start}}, \text{CALL}_{\text{start}}^{\text{skip}}$

These cases are analogous to the above one, where we additionally need $\dot{\omega}, \dot{\sigma} \models_G \neg z'.\text{started}$ and $\dot{\omega}, \dot{\sigma} \models_G z'.\text{started}$, respectively, to be able to apply the cooperation test. The above properties result from the transition antecedents $\neg\text{started}(\dot{T}, \beta)$ and $\text{started}(\dot{T}, \beta)$, respectively, using Lemma B.4 and $\dot{\omega}(z') = \beta$.

Case: $\text{CALL}_{\text{monitor}}$

As above, where $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} \text{thread}(z'.\text{lock}) = \text{thread}$ is implied by the transition antecedent $\text{owns}(\xi \downarrow \text{prog}, \beta)$ for the executing thread ξ , and Lemma B.2.

Case: RETURN

This case is analogous to the CALL case, where we define q_1 as the precondition of the corresponding return statement instead of the callee class invariant. The requirement $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$ of the cooperation test results from the fact that the values of formal parameters may not change during method execution, and that the method invocation statements may not contain instance variables, so that the values of the formal parameters and the expressions in the method invocation statement are untouched during the execution of the invoked method.

For the application of the interference freedom test, to show the validity of the interleavable predicate, we use the fact that the assertion $\text{pre}(stm_3)$ neither describes the caller nor the callee, since the corresponding local configuration is not involved in the execution.

Case: $\text{RETURN}_{\text{run}}$

Similar to the return case.

Case: $\text{RETURN}_{\text{wait}}$

In this case the antecedent $\neg \text{owns}(\dot{T} \downarrow \text{prog}, \beta)$ of the transition rule together with Lemma B.3 imply $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} z'.\text{lock} = \text{free}$. Furthermore, the executing thread is in the notified set prior to execution, and the same lemma yields that the executing thread is registered in $\dot{\sigma}(\beta)(\text{notified})$, i.e., $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} \text{thread}' \in z'.\text{notified}$.

Case: THROW_4

This case is similar to the RETURN case, where q_1 is the precondition of the given throw statement.

Case: TRY

Let the last computation step be the entering of a try-catch-finally block with observation $\vec{y} := \vec{e}$, executed in the local configuration $(\alpha, \dot{\tau}_1, \dot{stm}_1) \in \dot{T}$, resulting in $(\alpha, \dot{\tau}_1, \dot{stm}_1) \in \dot{T}$. According to the semantics, directly after entering the block but before the corresponding observation we have $\dot{\tau}_1 = \dot{\tau}_1[\text{exc} \mapsto \llbracket \text{exc} \rrbracket_{\mathcal{E}}^{\dot{\sigma}(\alpha), \dot{\tau}_1} \circ \text{null}]$ and $\dot{\sigma} = \dot{\sigma}$. After executing the observation we get $\dot{\tau}_1 = \dot{\tau}_1[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\dot{\sigma}(\alpha), \dot{\tau}_1}]$ and $\dot{\sigma} = \dot{\sigma}[\alpha.\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\dot{\sigma}(\alpha), \dot{\tau}_1}]$.

Since observations of try keywords must not change the values of variables occurring in GI , part (2) is satisfied.

For part (1), assume $(\beta, \tau_2, stm_2) \in \dot{T}$. If $(\beta, \tau_2, stm_2) = (\alpha, \dot{\tau}_1, \dot{stm}_1)$ is the executing local configuration, then by induction $\omega, \dot{\sigma}(\alpha), \dot{\tau}_1 \models_{\mathcal{L}} \text{pre}(\dot{stm}_1)$. The local correctness condition implies $\omega, \dot{\sigma}(\alpha), \dot{\tau}_1 \models_{\mathcal{L}} \text{pre}(\dot{stm}_1)[\vec{e}/\vec{y}][\text{exc} \circ \text{null}/\text{exc}]$. Using the properties of the local substitution formulated in Lemma 6.1 we get $\omega, \dot{\sigma}(\alpha), \dot{\tau}_1 \models_{\mathcal{L}} \text{pre}(\dot{stm}_1)$.

If otherwise (β, τ_2, stm_2) is not the executing local configuration, then it is contained in \dot{T} . If $\alpha \neq \beta$, i.e., the execution didn't take place in β , then $\dot{\sigma}(\beta) = \dot{\sigma}(\beta)$, and thus $\omega, \dot{\sigma}(\beta), \tau_2 \models_{\mathcal{L}} \text{pre}(stm_2)$ by induction. Otherwise, analogously to the argumentation above, the local correctness Condition 10 implies $\omega, \dot{\sigma}(\alpha), \dot{\tau}_1 \models_{\mathcal{L}} \text{pre}(\vec{y} := \vec{e})[\text{exc} \circ \text{null}/\text{exc}]$. Using the properties of the local substitution formulated in Lemma 6.1 we get $\omega, \dot{\sigma}(\alpha), \dot{\tau}_1 \models_{\mathcal{L}} \text{pre}(\vec{y} := \vec{e})$.

Let τ be $\dot{\tau}_1[\vec{v}' \mapsto \tau_2(\vec{v})]$, where $\vec{v} = \text{dom}(\tau_2)$ and \vec{v}' fresh. Then Lemma B.2, the induction assumptions, and the definition of interleavable imply

$$\omega, \dot{\sigma}(\alpha), \tau \models_{\mathcal{L}} \text{pre}(\vec{y} := \vec{e}) \wedge \text{pre}'(stm_2) \wedge \text{interleavable}(\text{pre}(stm_2), \vec{y} := \vec{e}),$$

and with the interference freedom test we get $\omega, \delta(\alpha), \tau \models_{\mathcal{L}} pre'(stm_2)[\vec{e}/\vec{y}]$. Using the substitution Lemma 6.1 and the fact that, due to the renaming mechanism, no variables in \vec{v}' may occur in \vec{y} , yields $\omega, \delta(\alpha), \tau_2 \models_{\mathcal{L}} pre(stm_2)$.

Part (3) is similar, using the fact that the class invariant may contain instance variables only, and thus its evaluation doesn't depend on the local state.

Case: FINALLY, YRT

These cases are analogous to the above one, where for FINALLY we have $\tilde{\tau}_1 = \tilde{\tau}_1$, and for YRT $\tilde{\tau}_1 = \tilde{\tau}_1[\text{exc}, \text{top} \mapsto \llbracket \text{head}(\text{exc}) \rrbracket_{\mathcal{E}}^{\delta(\alpha), \tilde{\tau}_1}, \llbracket \text{tail}(\text{exc}) \rrbracket_{\mathcal{E}}^{\delta(\alpha), \tilde{\tau}_1}]$; the substitution $[\text{exc} \circ \text{null}/\text{exc}]$ is replaced accordingly.

Case: THROW₁

Let $(\alpha, \tilde{\tau}, \hat{stm}) \in \hat{T}$ with $\hat{stm} = \text{throw } e; \langle \vec{y} := \vec{e} \rangle^{\text{throw}} stm_0; \text{catch } (c_1 u_1) stm_1 \dots; \text{catch } (c_n u_n) stm_n$ finally stm_{n+1} yrt; stm_{n+2} be the executing local configuration prior to the computation step, resulting in $(\alpha, \hat{\tau}, \hat{stm}) \in \hat{T}$ with $\hat{stm} = stm_i$; finally stm_{n+1} yrt; stm_{n+2} after execution. According to the semantics, $\llbracket e \rrbracket_{\mathcal{E}}^{\delta(\alpha), \tilde{\tau}} \in Val^{c_i}$ for some $1 \leq i \leq n$, implying $\llbracket e \neq \text{null} \wedge \text{hastype}(e, c_i) \rrbracket_{\mathcal{E}}^{\delta(\alpha), \tilde{\tau}}$. Furthermore, from $\forall 1 \leq j < i. \llbracket e \rrbracket_{\mathcal{E}}^{\delta(\alpha), \tilde{\tau}} \notin Val^{c_j}$ we conclude $\llbracket \forall 1 \leq j < i. \neg \text{hastype}(e, c_j) \rrbracket_{\mathcal{E}}^{\delta(\alpha), \tilde{\tau}}$.

Directly after exception throwing we have $\tilde{\tau} = \tilde{\tau}[u_i \mapsto \llbracket e \rrbracket_{\mathcal{E}}^{\delta(\alpha), \tilde{\tau}}]$ and $\delta = \delta$. The observation modifies the states resulting in $\hat{\tau} = \tilde{\tau}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\delta(\alpha), \tilde{\tau}}]$ and $\delta = \delta[\alpha. \vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\delta(\alpha), \tilde{\tau}}]$.

Since observations of exception throwing inside try-catch-finally blocks must not change the values of variables occurring in GI , part (2) is satisfied.

For part (1), assume $(\beta, \tau', \hat{stm}') \in \hat{T}$. If $(\beta, \tau', \hat{stm}') = (\alpha, \hat{\tau}, \hat{stm})$ is the executing local configuration, then by induction $\omega, \delta(\alpha), \tilde{\tau} \models_{\mathcal{L}} pre(\hat{stm})$. The local correctness Condition 17 implies $\omega, \delta(\alpha), \tilde{\tau} \models_{\mathcal{L}} pre(\hat{stm})[\vec{e}/\vec{y}][e/u_i]$. Using the properties of the local substitution formulated in Lemma 6.1 we get $\omega, \delta(\alpha), \hat{\tau} \models_{\mathcal{L}} pre(\hat{stm}')$.

If otherwise $(\beta, \tau', \hat{stm}')$ is not the executing local configuration, then it is contained in \hat{T} . If $\alpha \neq \beta$, i.e., the execution didn't take place in β , then $\delta(\beta) = \delta(\beta)$, and thus $\omega, \delta(\beta), \tau' \models_{\mathcal{L}} pre(\hat{stm}')$ by induction. Otherwise, the induction assumptions, the local correctness Condition 16, and the local substitution Lemma 6.1 imply $\omega, \delta(\alpha), \tilde{\tau}_1 \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$.

Let τ be $\tilde{\tau}[\vec{v}' \mapsto \tau'(\vec{v})]$, where $\vec{v} = \text{dom}(\tau')$ and \vec{v}' fresh. Then Lemma B.2, the induction assumptions, and the definition of interleavable imply

$$\omega, \delta(\alpha), \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e}) \wedge pre'(stm') \wedge \text{interleavable}(pre(stm'), \vec{y} := \vec{e}),$$

and with the interference freedom test we get $\omega, \delta(\alpha), \tau \models_{\mathcal{L}} pre'(stm')[\vec{e}/\vec{y}]$. Using the substitution Lemma 6.1 and the fact that, due to the renaming mechanism, no variables in \vec{v}' may occur in \vec{y} , yields $\omega, \delta(\alpha), \tau' \models_{\mathcal{L}} pre(stm')$.

Part (3) is similar, using the fact that the class invariant may contain instance variables only, and thus its evaluation doesn't depend on the local state.

Case: THROW₂, THROW₃, THROW₅

These cases are similar to the above one. None of these statements may change the values of variables occurring in the global invariant, and thus part (2) is satisfied.

The induction assumptions and the semantics assures that the antecedents of the corresponding local conditions hold in the configuration prior to execution. Satisfaction of the local conditions and the local

substitution lemma imply that the precondition of the statement of the executing local configuration hold after the computation step.

For the other local configurations, local correctness assures additionally, that the precondition of the attached observation hold directly before its execution. Again, we use induction assumption, satisfaction of the interference freedom conditions, and the local substitution lemma to show that the given assertion attached to the control point of the non-executing local configuration hold after observation.

Case: NEW

Let $(\alpha, \hat{\tau}_1, u := \text{new}; \langle \vec{y} := \vec{e} \rangle^{\text{new}} \text{stm}_1) \in \hat{T}$ be the local configuration of the executing thread prior to object creation, and $(\alpha, \hat{\tau}_1, \text{stm}_1) \in \hat{T}$ after it. Object creation updates the global state to $\check{\sigma} = \check{\sigma}[\beta \mapsto \sigma_{\text{inst}}^{\text{init}}[\text{this} \mapsto \beta]]$, where $\beta \notin \text{dom}(\check{\sigma})$; the executing thread's local state gets updated to $\check{\tau}_1 = \hat{\tau}_1[u \mapsto \beta]$. After observation we have $\hat{\tau}_1 = \check{\tau}_1[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$ and for the global state $\check{\sigma} = \check{\sigma}[\alpha. \vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$.

In the following let $p_1 = \text{pre}(u := \text{new})$, $p_2 = \text{pre}(\vec{y} := \vec{e})$, and $p_3 = \text{post}(\vec{y} := \vec{e})$. By induction $\omega, \check{\sigma} \models_{\mathcal{G}} GI$ and $\omega, \check{\sigma}(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} p_1$. Using the lifting lemma we get $\hat{\omega}, \check{\sigma} \models_{\mathcal{G}} GI \wedge P_1(z)$ for $\hat{\omega} = \omega[z \mapsto \alpha][\vec{v}_1 \mapsto \hat{\tau}_1(\vec{v}_1)]$ and \vec{v}_1 the variables from the domain of $\hat{\tau}_1$. Lemma 2.2 yields $\hat{\omega}[z' \mapsto \text{dom}(\check{\sigma})][u \mapsto \beta], \check{\sigma} \models_{\mathcal{G}} (GI \wedge (\exists u. P_1(z))) \downarrow z'$. Note that GI may not contain free logical variables, and thus its evaluation does not depend on the logical environment. The newly created object with a fresh identity is in its initial instance state, implying $\hat{\omega}[z' \mapsto \text{dom}(\check{\sigma})][u \mapsto \beta], \check{\sigma} \models_{\mathcal{G}} \text{Fresh}(z', u)$. Thus the cooperation test for object creation implies

$$\hat{\omega}[u \mapsto \beta], \check{\sigma} \models_{\mathcal{G}} P_2(z) \wedge I_{\text{new}}(u) \wedge (GI \wedge P_3(z))[\vec{E}(z)/z. \vec{y}],$$

where I_{new} is the class invariant of the new object. Using the lifting lemma again but in the reverse direction and Lemma 6.2 results $\omega, \check{\sigma} \models_{\mathcal{G}} GI$, and thus part (2). Note that in the annotation no free logical variables occur, and thus the values of assertions do not depend on the logical environment.

Furthermore, using the substitution lemmas we get

$$\omega, \check{\sigma}(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} p_2, \quad \omega, \check{\sigma}(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} p_3, \quad \text{and} \quad \omega, \check{\sigma}(\beta), \tau \models_{\mathcal{L}} I_{\text{new}}$$

for all τ . For the class invariant of the executing thread, the interference freedom test implies $\omega, \check{\sigma}(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} I$, where I is the class invariant of α . Since I may contain instance variables only, its evaluation doesn't depend on the local state, and the required property holds. The state of other objects not involved in the last step is not changed in the last computation step, and part (3) is satisfied.

Furthermore, part (1) is satisfied for the local configuration involved in the last computation step. All other configurations $(\gamma, \tau_2, \text{stm}_2)$ in \hat{T} are also in \hat{T} and $\gamma \neq \beta$. If $\gamma \neq \alpha$, then $\check{\sigma}(\gamma) = \check{\sigma}(\gamma)$, and thus $\omega, \check{\sigma}(\gamma), \tau_2 \models_{\mathcal{L}} \text{pre}(\text{stm}_2)$ by induction.

Assume now $\gamma = \alpha$, and let τ be $\hat{\tau}_1[\vec{v}' \mapsto \tau_2(\vec{v})]$, where $\vec{v} = \text{dom}(\tau_2)$. Then, since $\check{\sigma}(\alpha) = \check{\sigma}(\alpha)$, Lemma B.2, the induction assumptions, and the definition of interleavable imply using the interference freedom test that $\omega, \check{\sigma}(\alpha), \tau \models_{\mathcal{L}} \text{pre}'(\text{stm}_2)[\vec{e}/\vec{y}]$. The substitution Lemma 6.1 and the fact that, due to the renaming mechanism, no local variables in \vec{v}' occur in \vec{y} , yields $\omega, \check{\sigma}(\alpha), \tau_2 \models_{\mathcal{L}} \text{pre}(\text{stm}_2)$. I.e., part (1) is satisfied. \square

Corollary B.1. If $\text{prog}' \vdash \varphi_{\text{prog}'}$ and $\models \varphi_{\text{prog}'} \rightarrow \varphi_{\text{prog}}$, then $\text{prog} \models \varphi_{\text{prog}}$.

Proof of the soundness Corollary B.1: The proof is straightforward using the soundness Lemma 7.1. \square

C. Completeness proof

The following lemma states that the variable `loc` indeed stores the current control point of a thread:

Lemma C.1. Let $\langle T, \sigma \rangle$ be a reachable configuration of $prog'$ and $(\alpha, \tau, stm) \in T$. Then $\tau(\text{loc}) \equiv stm$.

Proof of Lemma C.1: Straightforward by the definition of augmentation. \square

Proof of the local merging Lemma 7.1 on page 432: Assume two computations $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ and $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \hat{T}_2, \hat{\sigma}_2 \rangle$ of $prog'$, and let $(\alpha, \tau, stm) \in \hat{T}_1$ with $\alpha \in \text{dom}(\hat{\sigma}_1) \cap \text{dom}(\hat{\sigma}_2)$ and $\hat{\sigma}_1(\alpha)(h_{inst}) = \hat{\sigma}_2(\alpha)(h_{inst})$. We prove $(\alpha, \tau, stm) \in \hat{T}_2$ by induction over the sum of the length of the computations.

In the initial case both \hat{T}_1 and \hat{T}_2 contain the same single initial local configuration, and thus the property holds.

For the inductive case, let $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ and $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$ be the last steps of the computations. The augmentation definition implies that each computation step appends at most one element to the instance history of α . If $\hat{\sigma}_1(\alpha)(h_{inst}) = \hat{\sigma}_1(\alpha)(h_{inst})$, then, by the definition of the augmentation, $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ did not execute in α , i.e., $(\alpha, \tau, stm) \in \hat{T}_1$, and the property follows by induction. The case for $\hat{\sigma}_2(\alpha)(h_{inst}) = \hat{\sigma}_2(\alpha)(h_{inst})$ is analogous. Thus assume in the following $\hat{\sigma}_1(\alpha)(h_{inst}) = \hat{\sigma}_1(\alpha)(h_{inst}) \circ (\sigma_{inst}^1, \tau_1)$ and $\hat{\sigma}_2(\alpha)(h_{inst}) = \hat{\sigma}_2(\alpha)(h_{inst}) \circ (\sigma_{inst}^2, \tau_2)$. From $\hat{\sigma}_1(\alpha)(h_{inst}) = \hat{\sigma}_2(\alpha)(h_{inst})$ we conclude that $\hat{\sigma}_1(\alpha)(h_{inst}) = \hat{\sigma}_2(\alpha)(h_{inst})$ and $(\sigma_{inst}^1, \tau_1) = (\sigma_{inst}^2, \tau_2)$.

Since $\hat{\sigma}_1(\alpha)(h_{inst}) \neq \hat{\sigma}_1(\alpha)(h_{inst})$, the computation step $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ executed some statements in α . If there is only one local configuration in α that was involved in the step, then the augmentation definition and the local substitution lemma imply that its resulting local configuration in \hat{T}_1 is given by (α, τ_1, stm_1) with $stm_1 \equiv \tau_1(\text{loc})$. From $(\sigma_{inst}^1, \tau_1) = (\sigma_{inst}^2, \tau_2)$ we conclude that the same local configuration executed in $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$. Thus, either $(\alpha, \tau, stm) \in \hat{T}_1$ is the executing configuration (α, τ_1, stm_1) and then it is also in \hat{T}_2 , or not, and then it is in \hat{T}_1 , by induction in \hat{T}_2 , and since it wasn't involved in the execution $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$, also in \hat{T}_2 .

If otherwise there are two local configurations in α involved in $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$, then $(\sigma_{inst}^1, \tau_1)$ specifies the callee's instance local state. However, due to the built-in auxiliary variables, the identity of the caller local configuration is also stored in τ_1 , in the formal parameter caller of the callee. The caller configuration is in \hat{T}_1 , and by induction in \hat{T}_2 . Furthermore, since there are no two local configurations with the same identity in a reachable configuration, both steps execute in the same instance local configuration.

Thus, either $(\alpha, \tau, stm) \in \hat{T}_1$ is one of the executing configurations and then it is also in \hat{T}_2 , or not, and then it is in \hat{T}_1 , by induction in \hat{T}_2 , and since it wasn't involved in the execution, also in \hat{T}_2 . \square

Proof of the global merging Lemma 7.2 on page 432: Assume two reachable configurations $\langle \hat{T}_1, \hat{\sigma}_1 \rangle$ and $\langle \hat{T}_2, \hat{\sigma}_2 \rangle$ and let $\alpha \in \text{dom}(\hat{\sigma}_1) \cap \text{dom}(\hat{\sigma}_2)$ satisfying $\hat{\sigma}_1(\alpha)(h_{comm}) = \hat{\sigma}_2(\alpha)(h_{comm})$. We show that there exists a reachable $\langle \hat{T}, \hat{\sigma} \rangle$ with $\text{dom}(\hat{\sigma}) = \text{dom}(\hat{\sigma}_2)$, $\hat{\sigma}(\alpha) = \hat{\sigma}_1(\alpha)$, and $\hat{\sigma}(\beta) = \hat{\sigma}_2(\beta)$ for all $\beta \in \text{dom}(\hat{\sigma}_2) \setminus \{\alpha\}$. We proceed by induction on the sum of the lengths of the computations.

In the base case we are given $\langle \hat{T}_1, \hat{\sigma}_1 \rangle = \langle \hat{T}_2, \hat{\sigma}_2 \rangle$ and the property trivially holds.

For the inductive step, let $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ and $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$ be the last steps of the computations.

If $\alpha \notin \text{dom}(\hat{\sigma}_1)$ or $\alpha \notin \text{dom}(\hat{\sigma}_2)$, then α was created in one of the last steps, and thus $\hat{\sigma}_1(\alpha)(\mathbf{h}_{\text{comm}}) = \hat{\sigma}_2(\alpha)(\mathbf{h}_{\text{comm}}) = \epsilon$. That means, no methods of α were involved yet, i.e., α is in its initial instance state $\hat{\sigma}_1(\alpha) = \hat{\sigma}_2(\alpha) = \sigma_{\text{inst}}^{\text{init}}[\text{this} \mapsto \alpha]$; in this case $\langle \hat{T}_2, \hat{\sigma}_2 \rangle$ already satisfies the requirements. Assume in the following $\alpha \in \text{dom}(\hat{\sigma}_1) \cap \text{dom}(\hat{\sigma}_2)$. We distinguish whether the last computation steps update the communication history of α or not.

Case: $\hat{\sigma}_1(\alpha)(\mathbf{h}_{\text{comm}}) = \hat{\sigma}_1(\alpha)(\mathbf{h}_{\text{comm}})$

In this case $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ doesn't execute any non-self communication or object creation in α . By induction there is a computation $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \hat{T}, \hat{\sigma} \rangle$ leading to a configuration such that $\hat{\sigma}(\alpha) = \hat{\sigma}_1(\alpha)$ and $\hat{\sigma}(\beta) = \hat{\sigma}_2(\beta)$ for all $\beta \in \text{dom}(\hat{\sigma}_2) \setminus \{\alpha\}$.

In case $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ does not execute in α at all, i.e., $\hat{\sigma}_1(\alpha) = \hat{\sigma}_1(\alpha)$, then $\langle \hat{T}, \hat{\sigma} \rangle$ already satisfies the requirements.

Otherwise, the local configurations in \hat{T}_1 which execute in α and which are involved in the computation step $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ are by the local merging Lemma 7.1 also in \hat{T} . Furthermore, from $\hat{\sigma}_1(\alpha)(\mathbf{h}_{\text{comm}}) = \hat{\sigma}_1(\alpha)(\mathbf{h}_{\text{comm}})$ we conclude that they don't execute any non-self communication or object creation, and thus their enabledness and effect depends only on the instance state of α . That means, the same computation as in $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ can be executed in $\langle \hat{T}, \hat{\sigma} \rangle$, leading to a reachable global configuration satisfying the requirements.

Case: $\hat{\sigma}_2(\alpha)(\mathbf{h}_{\text{comm}}) = \hat{\sigma}_2(\alpha)(\mathbf{h}_{\text{comm}})$

In this case $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$ does not execute any non-self communication or object creation involving α . By induction, there is a reachable $\langle \hat{T}, \hat{\sigma} \rangle$ with $\hat{\sigma}(\alpha) = \hat{\sigma}_1(\alpha)$ and $\hat{\sigma}(\beta) = \hat{\sigma}_2(\beta)$ for all $\beta \in \text{dom}(\hat{\sigma}_2) \setminus \{\alpha\}$.

If $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$ performs a step within α , then, according to the case assumption, it executes exclusively within α . This means, $\hat{\sigma}_2(\beta) = \hat{\sigma}_2(\beta)$ for all $\beta \in \text{dom}(\hat{\sigma}_2) \setminus \{\alpha\}$, and $\langle \hat{T}, \hat{\sigma} \rangle$ already satisfies the required properties.

If otherwise $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$ does not execute in α , then all local configurations in \hat{T}_2 , executing in an object different from α , are also in \hat{T} ; this follows from $\hat{\sigma}_2(\beta) = \hat{\sigma}(\beta)$ for all $\beta \in \text{dom}(\hat{\sigma}_2) \setminus \{\alpha\}$, and with the help of the local merging Lemma 7.1 applied to $\langle \hat{T}, \hat{\sigma} \rangle$ and $\langle \hat{T}_2, \hat{\sigma}_2 \rangle$. The enabledness of local configurations, whose execution does not involve α , are independent of the instance state of α ; furthermore, the effect of their execution neither influences the instance state of α nor depends on it. Thus in $\langle \hat{T}, \hat{\sigma} \rangle$ we can execute the same computation steps as in $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$, leading to a reachable configuration with the required properties.

Case: $\hat{\sigma}_1(\alpha)(\mathbf{h}_{\text{comm}}) \neq \hat{\sigma}_1(\alpha)(\mathbf{h}_{\text{comm}})$ and $\hat{\sigma}_2(\alpha)(\mathbf{h}_{\text{comm}}) \neq \hat{\sigma}_2(\alpha)(\mathbf{h}_{\text{comm}})$

In this case finally both $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ and $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$ execute some object creation or non-self communication in α , including exception throwing between different objects. We show that in this case $\hat{\sigma}_1(\alpha)(\mathbf{h}_{\text{comm}}) = \hat{\sigma}_2(\alpha)(\mathbf{h}_{\text{comm}})$ implies also $\hat{\sigma}_1(\alpha)(\mathbf{h}_{\text{comm}}) = \hat{\sigma}_2(\alpha)(\mathbf{h}_{\text{comm}})$, and thus by induction there is a computation leading to a configuration $\langle \hat{T}, \hat{\sigma} \rangle$ such that $\text{dom}(\hat{\sigma}) = \text{dom}(\hat{\sigma}_2)$, $\hat{\sigma}(\alpha) = \hat{\sigma}_1(\alpha)$, and $\hat{\sigma}(\beta) = \hat{\sigma}_2(\beta)$ for all other objects $\beta \in \text{dom}(\hat{\sigma}_2) \setminus \{\alpha\}$.

Furthermore, combining those local configurations involved in $\langle \hat{T}_1, \hat{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \hat{\sigma}_1 \rangle$ which execute within α with those in $\langle \hat{T}_2, \hat{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \hat{\sigma}_2 \rangle$ which execute outside α , we can define a computation $\langle \hat{T}, \hat{\sigma} \rangle \longrightarrow \langle \hat{T}, \hat{\sigma} \rangle$ such that $\hat{\sigma}(\alpha) = \hat{\sigma}_1(\alpha)$ and $\hat{\sigma}(\beta) = \hat{\sigma}_2(\beta)$ for all other objects $\beta \in \text{dom}(\hat{\sigma}_2) \setminus \{\alpha\}$.

The case assumptions imply, that the last elements of the communication histories $\acute{\sigma}_1(\alpha)(h_{comm})$ and $\acute{\sigma}_2(\alpha)(h_{comm})$ were appended in the last computation steps; $\acute{\sigma}_1(\alpha)(h_{comm}) = \acute{\sigma}_2(\alpha)(h_{comm})$ imply that the last elements are equal.

According to the augmentation, each computation step extends the communication history of α with at most one element. Thus we get $\acute{\sigma}_1(\alpha)(h_{comm}) = \acute{\sigma}_2(\alpha)(h_{comm})$, and by induction there is a reachable $\langle \hat{T}, \acute{\sigma} \rangle$ with $dom(\acute{\sigma}) = dom(\acute{\sigma}_2)$, $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$, and $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \setminus \{\alpha\}$.

Note that the last elements of the communication histories $\acute{\sigma}_1(\alpha)(h_{comm})$ and $\acute{\sigma}_2(\alpha)(h_{comm})$ record the kind of execution, and so we know that both steps execute the same kind of communication in α . Furthermore, the last elements record also the identity of the local configuration executing in α , the communication partner of α , and the communicated values, which are consequently also equal.

We distinguish on the kind of the computation step $\langle \hat{T}_1, \acute{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \acute{\sigma}_1 \rangle$:

Subcase: NEW

In this case $\acute{\sigma}_1(\alpha)(h_{comm}) = \acute{\sigma}_1(\alpha)(h_{comm}) \circ (\alpha, null, (new^c \gamma, thread_\alpha))$, where $thread_\alpha$ is the identity of the creator thread as specified by its local variable `thread`, and γ is the newly created object.

From the preliminary observations we conclude that $\langle \hat{T}_2, \acute{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \acute{\sigma}_2 \rangle$ creates the same new object γ being in the same initial state; furthermore, it leaves the states of all objects from $dom(\acute{\sigma}_2) \setminus \{\alpha\}$ untouched.

As $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$, the local merging Lemma 7.1 implies that the local configuration of the creator in \hat{T}_1 is also contained in \hat{T} . Thus, since $\gamma \notin dom(\acute{\sigma}_2) = dom(\acute{\sigma})$, the same computation step as in $\langle \hat{T}_1, \acute{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \acute{\sigma}_1 \rangle$ can be executed also in $\langle \hat{T}, \acute{\sigma} \rangle$, leading to a reachable configuration $\langle \hat{T}, \acute{\sigma} \rangle$ with $Val^{Object}(\acute{\sigma}) = Val^{Object}(\acute{\sigma}) \dot{\cup} \{\gamma\} = Val^{Object}(\acute{\sigma}_2) \dot{\cup} \{\gamma\} = Val^{Object}(\acute{\sigma}_2)$, $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$, and $\acute{\sigma}(\beta) = \acute{\sigma}(\beta) = \acute{\sigma}_2(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \setminus \{\alpha\}$. Finally, for the newly created object we have $\acute{\sigma}(\gamma) = \acute{\sigma}_2(\gamma) = \sigma_{inst}^{init}[\text{this} \mapsto \gamma]$, and thus $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \setminus \{\alpha\}$.

Subcase: CALL

Assume first that α is the caller object and $\beta \neq \alpha$ the callee. According to the preliminary observations, also $\langle \hat{T}_2, \acute{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \acute{\sigma}_2 \rangle$ executes the invocation of the same method of β , where α is the caller and β the callee. Furthermore, by the local merging lemma, the caller local configuration from \hat{T}_1 is also in \hat{T} , and its execution is also enabled in $\langle \hat{T}, \acute{\sigma} \rangle$. The last property holds also for synchronized and monitor methods, since the invocation of the same method of β by the same thread is enabled in $\langle \hat{T}_2, \acute{\sigma}_2 \rangle$, and $\acute{\sigma}_2(\beta) = \acute{\sigma}(\beta)$.

Thus the caller local configuration from \hat{T}_1 can execute the method invocation in $\langle \hat{T}, \acute{\sigma} \rangle$, leading to a reachable configuration $\langle \hat{T}, \acute{\sigma} \rangle$ with $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$. Furthermore, $\langle \hat{T}, \acute{\sigma} \rangle \longrightarrow \langle \hat{T}, \acute{\sigma} \rangle$ and $\langle \hat{T}_2, \acute{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \acute{\sigma}_2 \rangle$ execute the same callee observation in the same instance state $\acute{\sigma}_2(\beta) = \acute{\sigma}(\beta)$ and the same initial local state after the communication of the same actual parameter values, and thus $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$. The states of other objects are not touched, and thus $\langle \hat{T}, \acute{\sigma} \rangle$ satisfies the required properties.

Similarly, if the callee object is α , then the same caller local configuration as in $\langle \hat{T}_2, \acute{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \acute{\sigma}_2 \rangle$ can execute in $\langle \hat{T}, \acute{\sigma} \rangle$ leading to a reachable configuration satisfying the requirements.

Subcase: RETURN, THROW₄

These cases are analogous to the above case for CALL. The computation $\langle \hat{T}, \acute{\sigma} \rangle \longrightarrow \langle \hat{T}, \acute{\sigma} \rangle$ is constructed from the execution of the local configuration in α which executes in $\langle \hat{T}_1, \acute{\sigma}_1 \rangle \longrightarrow \langle \hat{T}_1, \acute{\sigma}_1 \rangle$, together with the execution of the communication partner of α which executes in $\langle \hat{T}_2, \acute{\sigma}_2 \rangle \longrightarrow \langle \hat{T}_2, \acute{\sigma}_2 \rangle$.

□

Lemma C.2. (Initial correctness)

The proof outline $prog'$ satisfies the initial conditions of Definition 6.1.

Proof of Lemma C.2: Let $\{p_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{p_3\} stm$; return be the main statement with local variables \vec{v} , and let I be the class invariant of the main class. We have to show for arbitrary $\sigma \in \Sigma$ and $\omega \in \Omega$ referring only to values existing in σ , that

$$\omega, \sigma \models_{\mathcal{G}} \text{InitState}(z) \wedge (\forall z'. z' = \text{null} \vee z = z') \rightarrow \\ P_2(z) \circ f_{init} \wedge (GI \wedge P_3(z) \wedge I(z)) \circ f_{obs} \circ f_{init},$$

where z is of the type of the main class, z' of type Object, and where $f_{init} = [z, (\text{null}, 0, \text{null})/\text{thread}, \text{caller}][\text{Init}(\vec{v})/\vec{v}]$ and $f_{obs} = [\vec{E}_2(z)/z.\vec{y}_2]$. We observe that

$$\omega, \sigma \models_{\mathcal{G}} \text{InitState}(z) \wedge (\forall z'. z' = \text{null} \vee z' = z)$$

implies that σ is the initial global state prior to the execution of the callee observation at the beginning of the main statement, i.e., defining exactly one existing object $\omega(z) = \alpha$ being in its initial instance state $\sigma(\alpha) = \sigma_{inst}^{init}[\text{this} \mapsto \alpha]$. We start transforming the right-hand side using the substitution Lemmas 6.2 and 2.1:

$$\begin{aligned} & \llbracket P_2(z)[z, (\text{null}, 0, \text{null})/\text{thread}, \text{caller}][\text{Init}(\vec{v})/\vec{v}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} \\ &= \llbracket P_2(z)[z, (\text{null}, 0, \text{null})/\text{thread}, \text{caller}] \rrbracket_{\mathcal{G}}^{\omega[\vec{v} \mapsto \text{Init}(\vec{v})], \sigma} \\ &= \llbracket P_2(z) \rrbracket_{\mathcal{G}}^{\omega[\vec{v} \mapsto \text{Init}(\vec{v})][\text{thread} \mapsto \alpha], \sigma} \\ &= \llbracket p_2 \rrbracket_{\mathcal{L}}^{\omega, \sigma(\alpha), \tau} \end{aligned}$$

with τ defined by $\tau^{init}[\text{thread} \mapsto \alpha][\text{caller} \mapsto (\text{null}, 0, \text{null})]$. The above value is *true*, since the run-method of the main class is initially invoked in the given context.

For the global invariant we get similarly

$$\begin{aligned} & \llbracket GI[\vec{E}_2(z)/z.\vec{y}_2][z, (\text{null}, 0, \text{null})/\text{thread}, \text{caller}][\text{Init}(\vec{v})/\vec{v}] \rrbracket_{\mathcal{G}}^{\omega, \sigma} \\ &= \llbracket GI[\vec{E}_2(z)/z.\vec{y}_2] \rrbracket_{\mathcal{G}}^{\omega[\vec{v} \mapsto \text{Init}(\vec{v})][\text{thread} \mapsto \alpha], \sigma} \\ &= \llbracket GI \rrbracket_{\mathcal{G}}^{\omega', \sigma'} \\ &= \llbracket GI \rrbracket_{\mathcal{G}}^{\omega, \sigma'} \end{aligned}$$

for some logical environment ω' and σ' given by $\sigma[\alpha.\vec{y}_2 \mapsto \llbracket \vec{e}_2 \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau}]$. In the last step we used the restriction that the global invariant may not contain free logical variables. The step before made use of the following equation for $\vec{E}_2(z)$, which we get using Lemma 2.1 and with the fact that \vec{e}_2 does not contain logical variables:

$$\begin{aligned} \llbracket \vec{E}_2(z) \rrbracket_{\mathcal{G}}^{\omega[\vec{v} \mapsto \text{Init}(\vec{v})][\text{thread} \mapsto \alpha], \sigma} &= \llbracket \vec{e}_2[z/\text{this}] \rrbracket_{\mathcal{G}}^{\omega[\vec{v} \mapsto \text{Init}(\vec{v})][\text{thread} \mapsto \alpha], \sigma} \\ &= \llbracket \vec{e}_2 \rrbracket_{\mathcal{G}}^{\omega[\vec{v} \mapsto \text{Init}(\vec{v})][\text{thread} \mapsto \alpha], \sigma(\alpha), \tau} \\ &= \llbracket \vec{e}_2 \rrbracket_{\mathcal{G}}^{\omega', \sigma(\alpha), \tau}. \end{aligned}$$

Since $\langle T', \sigma' \rangle$ with $T' = \{(\alpha, \tau', stm)\}$ and $\tau' = \tau[\vec{y}_2 \mapsto \llbracket \vec{e}_2 \rrbracket_{\mathcal{E}}^{\sigma(\alpha), \tau}]$ is an initial global configuration of $prog'$ after the observation at the beginning of the main statement, it is reachable, and the initial condition for the global invariant is satisfied. The cases for p_3 and I are similar to that of GI , where we additionally use the lifting substitution Lemma 2.1 to show that $\llbracket P_3(z) \rrbracket_{\mathcal{G}}^{\omega', \sigma'} = \llbracket p_3 \rrbracket_{\mathcal{L}}^{\omega', \sigma'(\alpha), \tau'}$. \square

Lemma C.3. (Local correctness: Assignment)

The proof outline $prog'$ satisfies the conditions of local correctness from Definition 6.2.

Proof of Lemma C.3: Let c be a class of $prog'$ with class invariant I , $\omega \in \Omega$, $\sigma_{inst} \in \Sigma_{inst}$, and $\tau \in \Sigma_{loc}$ with $\sigma_{inst}(\text{this}) = \alpha$. Assume a multiple assignment $\{p_1\} \vec{y} := \vec{e}\{p_2\}$ in c which is not the observation of communication or object creation. We have to show that

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p_1 \rightarrow p_2[\vec{e}/\vec{y}].$$

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p_1$ it follows by the definition of the annotation that there is a reachable $\langle \hat{T}, \hat{\sigma} \rangle$ with $\hat{\sigma}(\alpha) = \sigma_{inst}$ and $(\alpha, \tau, \vec{y} := \vec{e}; stm) \in \hat{T}$. Executing the local configuration in $\langle \hat{T}, \hat{\sigma} \rangle$ leads to a reachable global configuration $\langle \hat{T}, \hat{\sigma} \rangle$ with $\hat{\sigma}(\alpha) = \sigma_{inst}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}]$ and $(\alpha, \tau[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}], stm) \in \hat{T}$. Thus by the definition of the annotation for $prog'$ we have

$$\omega, \sigma_{inst}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}] \models_{\mathcal{L}} p_2,$$

and further with the substitution Lemma 6.1 $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p_2[\vec{e}/\vec{y}]$, as required. \square

Lemma C.4. (Local correctness: Exception handling)

The proof outline $prog'$ satisfies the conditions of local correctness from Definition 6.3.

Proof of Lemma C.4: Let stm be a statement of the form $\text{try } \langle^{try} \vec{y}_{try} := \vec{e}_{try} stm_0; \text{catch } (c_1 u_1) stm_1 \dots; \text{catch } (c_n u_n) stm_n \text{ finally } \langle \vec{y}_{fin} := \vec{e}_{fin} \rangle^{fin} stm_{n+1} \text{ yrt } \langle \vec{y}_{yrt} := \vec{e}_{yrt} \rangle^{yrt}$ in a class c . We show that for all $\hat{\omega}$, $\hat{\sigma}_{inst}$, and $\hat{\tau}$,

$$\hat{\omega}, \hat{\sigma}_{inst}, \hat{\tau} \models_{\mathcal{L}} pre(stm) \rightarrow pre(\vec{y}_{try} := \vec{e}_{try})[\text{exc} \circ \text{null}/\text{exc}] \wedge pre(stm_0)[\vec{e}_{try}/\vec{y}_{try}][\text{exc} \circ \text{null}/\text{exc}].$$

From $\hat{\omega}, \hat{\sigma}_{inst}, \hat{\tau} \models_{\mathcal{L}} pre(stm)$ it follows by the definition of the annotation that there is a reachable $\langle \hat{T}, \hat{\sigma} \rangle$ with $\hat{\sigma}(\alpha) = \hat{\sigma}_{inst}$ and $(\alpha, \hat{\tau}, stm; stm') \in \hat{T}$. Executing the exception throwing in the above local configuration in $\langle \hat{T}, \hat{\sigma} \rangle$ updates the local state to $\hat{\tau} = \hat{\tau}[\text{exc} \mapsto \llbracket \text{exc} \rrbracket_{\mathcal{E}}^{\hat{\sigma}_{inst}, \hat{\tau}} \circ \text{null}]$. The corresponding observation completes the computation step and leads to a reachable global configuration $\langle \hat{T}, \hat{\sigma} \rangle$ with $\hat{\sigma} = \hat{\sigma}[\alpha. \hat{\sigma}_{inst}[\vec{y}_{try} \mapsto \llbracket \vec{e}_{try} \rrbracket_{\mathcal{E}}^{\hat{\sigma}_{inst}, \hat{\tau}}] \mapsto]$, $\hat{\tau} = \hat{\tau}[\vec{y}_{try} \mapsto \llbracket \vec{e}_{try} \rrbracket_{\mathcal{E}}^{\hat{\sigma}_{inst}, \hat{\tau}}]$, and $(\alpha, \hat{\tau}, stm_0; \text{catch } (c_1 u_1) stm_1 \dots; \text{catch } (c_n u_n) stm_n \text{ finally } \langle \vec{y}_{fin} := \vec{e}_{fin} \rangle^{fin} stm_{n+1} \text{ yrt}) \in \hat{T}$.

Thus by the definition of the annotation for $prog'$ we have

$$\hat{\omega}, \hat{\sigma}_{inst}, \hat{\tau} \models_{\mathcal{L}} pre(stm_0),$$

and further with the substitution Lemma 6.1

$$\hat{\omega}, \hat{\sigma}_{inst}, \hat{\tau} \models_{\mathcal{L}} pre(stm_0)[\vec{e}_{fin}/\vec{y}_{fin}][exc \circ null/exc].$$

Note that the annotation may not contain free logical variables.

The case for the precondition of the observation is similar: By definition we have $\hat{\omega}, \hat{\sigma}_{inst}, \hat{\tau} \models_{\mathcal{L}} pre(\vec{y}_{try} := \vec{e}_{try})$, and thus $\hat{\omega}, \hat{\sigma}_{inst}, \hat{\tau} \models_{\mathcal{L}} pre(\vec{y}_{try} := \vec{e}_{try})[exc \circ null/exc]$, as required.

The other cases are similar. The antecedents of the conditions assure reachability and enabledness; we use the local substitution lemma to show the required properties. \square

Lemma C.5. (Interference freedom)

The proof outline $prog'$ satisfies the conditions for interference freedom from Definition 6.4.

Proof of Lemma C.5: Assume an arbitrary assignment $\vec{y} := \vec{e}$ with precondition p in class c with class invariant I , and an arbitrary assertion q at a control point in the same class. We show the verification condition from Equation (32) on page 427

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p \wedge q' \wedge interleavable(q, \vec{y} := \vec{e}) \rightarrow q'[\vec{e}/\vec{y}],$$

for some logical environment ω together with some instance and local states σ_{inst} and τ , where q' denotes q with all local variables u replaced by some fresh local variables u' .

Let $\alpha = \sigma_{inst}(\text{this})$, and assume first that $\vec{y} := \vec{e}$ is not the observation of communication, object creation, or exception throwing or handling. The first clause $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ implies that there exists a computation reaching $\langle \hat{T}_p, \hat{\sigma}_p \rangle$ with $\hat{\sigma}_p(\alpha) = \sigma_{inst}$, and a configuration $(\alpha, \tau, \vec{y} := \vec{e}; stm'_p) \in \hat{T}_p$.

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'$ we get by renaming back the local variables that $\omega, \sigma_{inst}, \tau' \models_{\mathcal{L}} q$ for $\tau'(u) = \tau(u')$ for all local variables u in q . Let q be the precondition of the statement stm_q . Note that q is an assertion at a control point. Applying the annotation definition we conclude that there is a reachable $\langle \hat{T}_q, \hat{\sigma}_q \rangle$ with $\hat{\sigma}_q(\alpha) = \sigma_{inst} = \hat{\sigma}_p(\alpha)$ and $(\alpha, \tau', stm_q; stm'_q) \in \hat{T}_q$. The local merging Lemma 7.1 implies that $(\alpha, \tau', stm_q; stm'_q) \in \hat{T}_p$.

Let $\langle \hat{T}_p, \hat{\sigma}_p \rangle$ result from $\langle \hat{T}_p, \hat{\sigma}_p \rangle$ by executing the enabled local configuration $(\alpha, \tau, \vec{y} := \vec{e}; stm'_p)$. We have $\hat{\sigma}_p(\alpha) = \sigma_{inst}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}]$. From the assumption $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} interleavable(q, \vec{y} := \vec{e})$ we get that $(\alpha, \tau', stm_q; stm'_q)$ is not the executing configuration, and thus $(\alpha, \tau', stm_q; stm'_q) \in \hat{T}_p$.

According to the annotation definition $\omega, \sigma_{inst}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau' \models_{\mathcal{L}} q$, and after renaming the local variables of q also $\omega, \sigma_{inst}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau \models_{\mathcal{L}} q'$. Due to renaming, no local variables of q' occur in \vec{y} , implying

$$\omega, \sigma_{inst}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}] \models_{\mathcal{L}} q'.$$

Finally, by the substitution Lemma 6.1 we get $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$.

If the assignment observes object creation, communication, or exception throwing or handling, the proof is similar. For object creation, $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ implies that there exists a computation reaching $\langle \hat{T}_p, \hat{\sigma}_p \rangle$ with $\hat{\sigma}_p(\alpha) = \sigma_{inst}$, and an enabled configuration $(\alpha, \tau_p, stm_p; stm'_p) \in \hat{T}_p$, where stm_p is of the form $u := \text{new}; (\vec{y} := \vec{e})^{new}$. The local state τ_p is $\tau[u \mapsto v]$ for some value v , such that the local configuration is enabled to create $\tau(u)$. Directly after creation, the creator local configuration has

the local state τ and executes its observation resulting in the local state $\tau[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}]$ and instance state $\sigma_{inst}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\sigma_{inst}, \tau}]$. Note that σ_{inst} is not influenced by the object creation itself. Again, the interleavable predicate assures that $(\alpha, \tau', stm_q; stm'_q)$ is not the executing configuration, and we get $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$ as above.

The other cases for observations of communication, object creation, or exception throwing and handling are analogous. In the case of caller observation in a self-communication, the restrictions on the augmentation imply that $\vec{y} := \vec{e}$ does not change the values of instance variables, and the requirement follows directly from the assumptions. If p is the precondition of a callee observation at the beginning of a method body, then the annotation assure that the invocation of the method is enabled in $\langle \hat{T}_p, \delta_p \rangle$ such that τ is the local state of the callee directly after communication but before observation. Note that for self-communication, the caller part does not change the instance state. Thus the only update of the instance state of α is given by the effect of $\vec{y} := \vec{e}$. Again, the interleavable predicate assures that $(\alpha, \tau', stm_q; stm'_q)$ is neither the caller nor the callee, and thus $(\alpha, \tau', stm_q; stm'_q) \in \hat{T}_p$. We get $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$ as above.

Validity of the verification condition 31 for the class invariant is similar, where we additionally use the fact that the class invariant refers to instance variables only. \square

Lemma C.6. (Cooperation test: Communication)

The proof outline $prog'$ satisfies the verification conditions of the cooperation test for communication of Definition 6.5.

Proof of Lemma C.6: We distinguish on the kind of communication starting with the verification condition for synchronized method invocation.

Case: CALL

Let $\{p_1\} u_{ret} := e_0.m(\vec{e}); \{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_3\}^{wait}$ be a statement in a class c of $prog'$ with e_0 of type c' , where method $m \notin \{\text{start, wait, notify, notifyAll}\}$ of c' is synchronized with body $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\} stm$, formal parameters \vec{u} , local variables without the formal parameters given by \vec{v} , and let $q_1 = I_{c'}$ be the callee class invariant. Assume

$$\dot{\omega}, \delta \models_{\mathcal{G}} GI \wedge P_1(z) \wedge Q_1(z') \wedge \text{comm} \wedge z \neq \text{null} \wedge z' \neq \text{null}$$

for distinct and fresh $z \in LVar^c$ and $z' \in LVar^{c'}$, and where comm is $E_0(z) = z' \wedge (z'.\text{lock} = \text{free} \vee \text{thread}(z'.\text{lock}) = \text{thread})$. Note that for completeness we don't need the information stored in the caller class invariant. By definition of the global invariant, the assumption $\dot{\omega}, \delta \models_{\mathcal{G}} GI$ implies that there exists a reachable $\langle T, \sigma \rangle$ with

$$\text{dom}(\delta) = \text{dom}(\sigma) \text{ and } \delta(\gamma)(h_{comm}) = \sigma(\gamma)(h_{comm}) \text{ for all } \gamma \in \text{dom}(\sigma).$$

Assuming $\dot{\omega}(z) = \alpha$ as caller identity, $\dot{\omega}, \delta \models_{\mathcal{G}} P_1(z)$ implies $\dot{\omega}, \delta(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} p_1$ by the substitution Lemma 2.1, for some local state $\hat{\tau}_1$ with $\hat{\tau}_1(u) = \dot{\omega}(u)$ for all local variables u occurring in p_1 . By the annotation definition there exists a reachable configuration $\langle T_1, \sigma_1 \rangle$ such that

$$\sigma_1(\alpha) = \delta(\alpha) \text{ and } (\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1) \in T_1.$$

Recall that $\sigma(\gamma)(\mathbf{h}_{comm}) = \delta(\gamma)(\mathbf{h}_{comm})$ for all $\gamma \in \text{dom}(\sigma)$, and especially for the caller $\sigma(\alpha)(\mathbf{h}_{comm}) = \delta(\alpha)(\mathbf{h}_{comm}) = \sigma_1(\alpha)(\mathbf{h}_{comm})$. Using the global merging Lemma 7.2 applied to $\langle T_1, \sigma_1 \rangle$ and $\langle T, \sigma \rangle$ we get that there is a reachable $\langle T', \sigma' \rangle$ with $\text{dom}(\sigma') = \text{dom}(\sigma)$ and

$$\sigma'(\alpha) = \sigma_1(\alpha) \text{ and } \sigma'(\gamma) = \sigma(\gamma) \text{ for all } \gamma \in \text{dom}(\sigma) \setminus \{\alpha\}.$$

Furthermore, $(\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1) \in T_1$, $\sigma_1(\alpha) = \sigma'(\alpha)$, and the local merging Lemma 7.1 implies that

$$(\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1) \in T'.$$

Let $\beta = \omega(z')$ be the callee object. In case of a self-call, i.e., for $\alpha = \beta$, we directly get that $\langle T'', \sigma'' \rangle = \langle T', \sigma' \rangle$ is a reachable configuration such that $\sigma''(\alpha) = \delta(\alpha)$, $\sigma''(\gamma)(\mathbf{h}_{comm}) = \delta(\gamma)(\mathbf{h}_{comm})$ for all $\gamma \in \text{dom}(\delta)$, and $(\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1) \in T''$.

Otherwise, the assumption $\dot{\omega}, \delta \models_{\mathcal{G}} I_{c'}(z')$ implies $\dot{\omega}, \delta(\beta), \tau_2 \models_{\mathcal{L}} I_{c'}$ for some local state τ_2 . Note that the class invariant contains instance variables, only. By definition of the class invariant, there is a reachable global configuration $\langle T_2, \sigma_2 \rangle$ such that

$$\sigma_2(\beta) = \delta(\beta).$$

We need to fall back upon the two merging lemmas once more to obtain a common reachable configuration: Analogously to the caller part, the global merging Lemma 7.2 applied to $\langle T_2, \sigma_2 \rangle$ and $\langle T', \sigma' \rangle$ yields that there is a reachable configuration $\langle T'', \sigma'' \rangle$ with $\text{dom}(\sigma'') = \text{dom}(\sigma')$ and

$$\sigma''(\beta) = \sigma_2(\beta) \text{ and } \sigma''(\gamma) = \sigma'(\gamma) \text{ for all } \gamma \in \text{dom}(\sigma') \setminus \{\beta\}.$$

Now, $(\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1) \in T'$, $\sigma''(\alpha) = \sigma'(\alpha)$, and the local merging Lemma 7.1 implies that the local configuration $(\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1)$ is in T'' .

Thus $\langle T'', \sigma'' \rangle$ is a reachable configuration with $\sigma''(\alpha) = \delta(\alpha)$, $\sigma''(\beta) = \delta(\beta)$, $\sigma''(\gamma)(\mathbf{h}_{comm}) = \delta(\gamma)(\mathbf{h}_{comm})$ for all $\gamma \in \text{dom}(\delta)$, and $(\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1) \in T''$.

With the antecedent $\dot{\omega}, \delta \models_{\mathcal{G}} z'.\text{lock} = \text{free} \vee \text{thread}(z'.\text{lock}) = \text{thread}$ of the cooperation test we get $\delta(\beta)(\text{lock}) = \text{free} \vee \text{thread}(\delta(\beta)(\text{lock})) = \hat{\tau}_1(\text{thread})$. With $\delta(\beta) = \sigma''(\beta)$ and Lemma B.3 we get $\neg \text{owns}(T'' \setminus \{\xi\}, \beta)$, where ξ is the stack with $(\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1)$ on top. Furthermore, $\dot{\omega}, \delta \models_{\mathcal{G}} \text{comm}$ implies $\dot{\omega}, \delta \models_{\mathcal{G}} E_0(z) = z'$, and by the lifting substitution lemma $\llbracket e_0 \rrbracket_{\mathcal{E}}^{\delta(\alpha), \hat{\tau}_1 \sigma''_{inst 1}} = \llbracket e_0 \rrbracket_{\mathcal{E}}^{\sigma''(\alpha), \hat{\tau}_1} = \omega(z') = \beta$. This means, the invocation of method m of β is enabled in the local configuration $(\alpha, \hat{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1)$ in $\langle T'', \sigma'' \rangle$.

The definition of the augmentation, and $\sigma''(\alpha) = \delta(\alpha)$ gives

$$\dot{\omega}, \delta(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} p_2,$$

which by the substitution Lemma 2.1 and with the definition of $\hat{\tau}_1$ yields $\dot{\omega}, \delta \models_{\mathcal{G}} P_2(z)$. Due to the renaming mechanism we get

$$\dot{\omega}, \delta \models_{\mathcal{G}} P_2(z) \circ f_{comm}$$

for $f_{comm} = [\vec{E}(z), \text{Init}(\vec{v})/\vec{u}', \vec{v}']$. For the precondition of the method body, the annotation definition implies

$$\dot{\omega}, \delta(\beta), \hat{\tau}_2 \models_{\mathcal{L}} q_2$$

with $\tilde{\tau}_2 = \tau^{init}[\vec{u} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\tilde{\sigma}(\alpha), \tilde{\tau}_1}]$. For the actual parameters we obtain by the substitution Lemma 2.1 $\llbracket \vec{E}(z) \rrbracket_{\mathcal{G}}^{\dot{\omega}, \dot{\sigma}} = \llbracket \vec{e} \rrbracket_{\mathcal{L}}^{\dot{\omega}, \dot{\sigma}(\alpha), \tilde{\tau}_1} = \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\tilde{\sigma}(\alpha), \tilde{\tau}_1}$, and further with the same lemma

$$\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} Q'_2(z')[\vec{E}(z), \text{Init}(\vec{v})/\vec{u}', \vec{v}']$$

as required by the cooperation test.

Directly after communication we have a global configuration with still the same global state σ'' . The caller observation evolves its own local state to $\tilde{\tau}_1 = \tilde{\tau}_1[\vec{y}_1 \mapsto \llbracket \vec{e}_1 \rrbracket_{\mathcal{E}}^{\sigma''(\alpha), \tilde{\tau}_1}]$, and the global state to $\check{\sigma} = \sigma''[\alpha.\vec{y}_1 \mapsto \llbracket \vec{e}_1 \rrbracket_{\mathcal{E}}^{\sigma''(\alpha), \tilde{\tau}_1}]$. Finally, the callee observation changes the global state to $\dot{\sigma} = \check{\sigma}[\beta.\vec{y}_2 \mapsto \llbracket \vec{e}_2 \rrbracket_{\mathcal{E}}^{\check{\sigma}(\beta), \tilde{\tau}_2}]$, where its own local state is updated to $\tilde{\tau}_2 = \tilde{\tau}_2[\vec{y}_2 \mapsto \llbracket \vec{e}_2 \rrbracket_{\mathcal{E}}^{\check{\sigma}(\beta), \tilde{\tau}_2}]$. According to the annotation definition we get

$$\dot{\omega}, \dot{\sigma}(\alpha), \tilde{\tau}_1 \models_{\mathcal{L}} p_3, \quad \dot{\omega}, \dot{\sigma}(\beta), \tilde{\tau}_2 \models_{\mathcal{L}} q_3, \quad \text{and} \quad \dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} GI.$$

Let $\dot{\omega} = \dot{\omega}[\vec{v}' \mapsto \text{Init}(\vec{v})][\vec{u}' \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\dot{\omega}(\alpha), \tilde{\tau}_1}][\vec{y}_1 \mapsto \llbracket \vec{e}_1 \rrbracket_{\mathcal{E}}^{\dot{\omega}(\alpha), \tilde{\tau}_1}][\vec{y}_2 \mapsto \llbracket \vec{e}_2 \rrbracket_{\mathcal{E}}^{\dot{\omega}(\beta), \tilde{\tau}_2}]$. The lifting lemma implies $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} GI \wedge P_3(z) \wedge Q'_3(z')$; with the global substitution lemma finally

$$\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z) \wedge Q'_3(z'))[\vec{E}'_2(z')/z'.\vec{y}'_2][\vec{E}'_1(z)/z'.\vec{y}'_1][\vec{E}(z), \text{Init}(\vec{v})/\vec{u}', \vec{v}'],$$

and thus the cooperation test is satisfied for the invocation of synchronous methods.

The case for non-synchronized methods is analogous, where the antecedent $z'.\text{lock} = \text{free} \vee \text{thread}(z'.\text{lock}) = \text{thread}$ is dropped.

Case: CALL_{monitor}

This case is similar to the above one of CALL, where for the invocation of a method $m \in \{\text{wait}, \text{notify}, \text{notifyAll}\}$, the assertion comm is given by $E_0(z) = z' \wedge \text{thread}(z'.\text{lock}) = \text{thread}$, implying $\text{owns}(\xi, \beta)$ for the caller thread ξ and the callee object β .

Case: CALL_{start}

Enabledness of starting the thread of an object β requires $\neg \text{started}(T'', \beta)$. Due to the definition of comm, we have additionally $\dot{\omega}, \sigma'' \models_{\mathcal{G}} \neg z'.\text{started}$, which implies $\neg \sigma''(\beta)(\text{started})$. We get enabledness by Lemma B.4.

Case: CALL_{start}^{skip}

The enabledness argument is similar for CALL_{start}^{skip}, where we use $\dot{\omega}, \sigma'' \models_{\mathcal{G}} z'.\text{started}$ to imply the enabledness predicate $\text{started}(T'', \beta)$.

Case: RETURN

For return, the construction of $\langle T'', \sigma'' \rangle$ is similar, where we get instead of the enabledness of the caller that the callee configuration $(\beta, \tilde{\tau}_2, \text{return } e_{ret}; \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret})$ is in $\langle T'', \sigma'' \rangle$, and thus enabled to execute.

Case: RETURN_{wait}

In this case we additionally have to show $\neg \text{owns}(T'', \beta)$, which we get from the comm assertion implying $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} z'.\text{lock} = \text{free}$ and using Lemma B.3.

Case: RETURN_{run}

Since the run-method cannot be invoked directly, we conclude that the executing local configuration is the only one in its stack, i.e., the transition rule RETURN_{run} of the semantics can be applied in $\langle T'', \sigma'' \rangle$ to terminate the callee $(\beta, \tilde{\tau}_2, \text{return}; \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret})$.

□

Lemma C.7. (Cooperation test: Instantiation)

The proof outline $prog'$ satisfies the verification conditions of the cooperation test for object creation of Definition 6.6.

Proof of Lemma C.7: Let $\{p_1\} u := new^c; \{p_2\}^{new} \langle \vec{y} := \vec{e} \rangle^{new} \{p_3\}$ be a statement in class c' of $prog'$, and assume

$$\tilde{\omega}, \tilde{\sigma} \models_{\mathcal{G}} z \neq \text{null} \wedge z \neq u \wedge \exists z'. \text{Fresh}(z', u) \wedge (GI \wedge \exists u. P_1(z)) \downarrow z'$$

with $z \in LVar^{c'}$ and $z' \in LVar^{\text{list Object}}$ fresh. Note that we don't need the class invariant of the creator for completeness. We show that

$$\tilde{\omega}, \tilde{\sigma} \models_{\mathcal{G}} P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z))[\vec{E}(z)/z.\vec{y}].$$

Let $\tilde{\omega}(z) = \alpha$ and $\tilde{\omega}(u) = \beta$. According to the semantics of assertions we have that

$$\omega, \tilde{\sigma} \models_{\mathcal{G}} \text{Fresh}(z', u) \wedge (GI \wedge \exists u. P_1(z)) \downarrow z'$$

for some logical environment ω that assigns to z' a sequence of objects from $Val_{\text{null}}^{\text{Object}}(\tilde{\sigma}) = \bigcup_c Val_{\text{null}}^c(\tilde{\sigma})$, and agrees on the values of all other variables with $\tilde{\omega}$. The assertion $\text{Fresh}(z', u)$ is defined by

$$\text{InitState}(u) \wedge u \notin z' \wedge \forall v. v \in z' \vee v = u,$$

where $\text{InitState}(u)$ expands to $u \neq \text{null} \wedge \bigwedge_{x \in IVar_c} u.x = \text{init}(x)$. Thus, $\omega, \tilde{\sigma} \models_{\mathcal{G}} \text{Fresh}(z', u)$ implies that $\beta \in Val^c(\tilde{\sigma})$ with $\tilde{\sigma}(\beta) = \sigma_{\text{inst}}^{\text{init}}[\text{this} \mapsto \beta]$, and additionally $Val_{\text{null}}^{\text{Object}}(\tilde{\sigma}) = \omega(z') \dot{\cup} \{\beta\}$. Let $\tilde{\sigma}$ be the global state with domain $Val^{\text{Object}}(\tilde{\sigma}) = Val^{\text{Object}}(\tilde{\sigma}) \setminus \{\beta\}$ and such that $\tilde{\sigma}(\gamma) = \tilde{\sigma}(\gamma)$ for all objects $\gamma \in Val^{\text{Object}}(\tilde{\sigma})$. Then $\tilde{\sigma} = \tilde{\sigma}[\beta \mapsto \sigma_{\text{inst}}^{\text{init}}[\text{this} \mapsto \beta]]$, and from

$$\omega, \tilde{\sigma} \models_{\mathcal{G}} (GI \wedge \exists u. P_1(z)) \downarrow z'$$

we get with Lemma 2.2

$$\omega, \tilde{\sigma} \models_{\mathcal{G}} GI \wedge \exists u. P_1(z).$$

By definition of the annotation, $\omega, \tilde{\sigma} \models_{\mathcal{G}} GI$ implies that there is a reachable configuration $\langle \hat{T}_1, \tilde{\sigma}_1 \rangle$ such that

$$\text{dom}(\tilde{\sigma}_1) = \text{dom}(\tilde{\sigma}) \text{ and } \tilde{\sigma}_1(\gamma)(\mathbf{h}_{\text{comm}}) = \tilde{\sigma}(\gamma)(\mathbf{h}_{\text{comm}}) \text{ for all } \gamma \in \text{dom}(\tilde{\sigma}).$$

The precondition of the object creation statement

$$\omega, \tilde{\sigma} \models_{\mathcal{G}} \exists u. P_1(z)$$

implies

$$\omega[u \mapsto v], \tilde{\sigma} \models_{\mathcal{G}} P_1(z)$$

for some $v \in Val_{\text{null}}^{\text{Object}}(\tilde{\sigma})$. Applying the lifting Lemma 2.1 we get that

$$\omega, \tilde{\sigma}(\alpha), \hat{\tau} \models_{\mathcal{L}} p_1$$

for a local state $\hat{\tau}$ with $\hat{\tau}(u) = v$ and $\hat{\tau}(v) = \omega(v)$ for all other local variables v . By definition of the annotation, there is a reachable global configuration $\langle \hat{T}_2, \hat{\sigma}_2 \rangle$ such that

$$\hat{\sigma}_2(\alpha) = \hat{\sigma}(\alpha) \text{ and } (\alpha, \hat{\tau}, u := \text{new}^c; \langle \vec{y} := \vec{e} \rangle^{\text{new}} \text{ stm}) \in \hat{T}_2.$$

Recall that $\hat{\sigma}_1(\gamma)(h_{\text{comm}}) = \hat{\sigma}(\gamma)(h_{\text{comm}})$ for all $\gamma \in \text{dom}(\hat{\sigma})$; especially we have $\hat{\sigma}_1(\alpha)(h_{\text{comm}}) = \hat{\sigma}(\alpha)(h_{\text{comm}}) = \hat{\sigma}_2(\alpha)(h_{\text{comm}})$. Using the global merging Lemma 7.2 applied to the reachable global configurations $\langle \hat{T}_2, \hat{\sigma}_2 \rangle$ and $\langle \hat{T}_1, \hat{\sigma}_1 \rangle$ we get that there is a reachable configuration $\langle \hat{T}_3, \hat{\sigma}_3 \rangle$ with

$$\text{dom}(\hat{\sigma}_3) = \text{dom}(\hat{\sigma}_1), \hat{\sigma}_3(\alpha) = \hat{\sigma}_2(\alpha), \text{ and } \hat{\sigma}_3(\gamma) = \hat{\sigma}_1(\gamma) \text{ for all } \gamma \in \text{dom}(\hat{\sigma}_1) \setminus \{\alpha\}.$$

Furthermore, $(\alpha, \hat{\tau}, u := \text{new}^c; \langle \vec{y} := \vec{e} \rangle^{\text{new}} \text{ stm}) \in \hat{T}_2$, $\hat{\sigma}_2(\alpha) = \hat{\sigma}_3(\alpha)$, and the local merging Lemma 7.1 implies that $(\alpha, \hat{\tau}, u := \text{new}^c; \langle \vec{y} := \vec{e} \rangle^{\text{new}} \text{ stm}) \in \hat{T}_3$.

So we know that $\langle \hat{T}_3, \hat{\sigma}_3 \rangle$ is a reachable configuration containing the local configuration $(\alpha, \hat{\tau}, u := \text{new}^c; \langle \vec{y} := \vec{e} \rangle^{\text{new}} \text{ stm}) \in \hat{T}_3$. With $\text{Val}^{\text{Object}}(\hat{\sigma}) = \text{Val}^{\text{Object}}(\hat{\sigma}) \setminus \{\beta\}$, $\text{dom}(\hat{\sigma}_1) = \text{dom}(\hat{\sigma})$, and $\text{dom}(\hat{\sigma}_3) = \text{dom}(\hat{\sigma}_1)$ we get that $\beta \notin \text{dom}(\hat{\sigma}_3)$, i.e., the local configuration is enabled to create the fresh object $\beta = \omega(u)$. With $\hat{\sigma}_3(\alpha) = \hat{\sigma}_2(\alpha) = \hat{\sigma}(\alpha)$ we get

$$\omega, \check{\sigma}(\alpha), \check{\tau} \models_{\mathcal{L}} p_2,$$

where $\check{\tau} = \hat{\tau}[u \mapsto \beta]$; with the lifting Lemma 2.1 together with the definition of $\hat{\tau}$ this means $\omega, \check{\sigma} \models_{\mathcal{G}} P_2(z)$, as required in the cooperation test.

Executing the instantiation in the local configuration $(\alpha, \hat{\tau}, u := \text{new}^c; \langle \vec{y} := \vec{e} \rangle^{\text{new}} \text{ stm})$ in $\langle \hat{T}_3, \hat{\sigma}_3 \rangle$, creating a new object $\beta \notin \text{dom}(\hat{\sigma}_3)$, results in $\langle \check{T}_3, \check{\sigma}_3 \rangle$ with $\check{\sigma}_3 = \hat{\sigma}_3[\beta \mapsto \sigma_{\text{inst}}^{\text{init}}[\text{this} \mapsto \beta]]$; executing the creator observation leads to a reachable $\langle \check{T}_3, \check{\sigma}_3 \rangle$ with $\check{\sigma}_3 = \hat{\sigma}_3[\alpha.\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\hat{\sigma}_3(\alpha), \check{\tau}}]$ and $(\alpha, \check{\tau}, \text{stm})$ in \check{T}_3 with $\check{\tau} = \hat{\tau}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\hat{\sigma}_3(\alpha), \check{\tau}}]$.

As $\langle \check{T}_3, \check{\sigma}_3 \rangle$ is reachable with $\check{\sigma}_3(\beta) = \sigma_{\text{inst}}^{\text{init}}[\text{this} \mapsto \beta] = \check{\sigma}(\beta)$ we know

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

As I_c may not contain local variables, applying the lifting Lemma 2.1 again with $\omega(u) = \beta$ yields the required condition $\check{\omega}, \check{\sigma} \models_{\mathcal{G}} I_c(u)$ for the class invariant. It remains to show that

$$\check{\omega}, \check{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z))[\vec{E}(z)/z.\vec{y}].$$

Applying the substitution Lemma 6.2 and the fact that GI does not contain free logical variables yields

$$\llbracket GI[\vec{E}(z)/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}} = \llbracket GI \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}}$$

with $\check{\sigma} = \hat{\sigma}[\alpha.\vec{y} \mapsto \llbracket \vec{E}(z) \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}}]$. Thus we have to show the existence of a reachable configuration with a global state defining the same object domain and communication history values as $\check{\sigma}$. The configuration $\langle \check{T}_3, \check{\sigma}_3 \rangle$ satisfies the above requirements, since, first, it is reachable with

$$\begin{aligned} \text{Val}^{\text{Object}}(\check{\sigma}_3) &= \text{Val}^{\text{Object}}(\hat{\sigma}_3) \cup \{\beta\} = \text{Val}^{\text{Object}}(\hat{\sigma}_1) \cup \{\beta\} \\ &= \text{Val}^{\text{Object}}(\hat{\sigma}) \cup \{\beta\} = \text{Val}^{\text{Object}}(\hat{\sigma}) = \text{Val}^{\text{Object}}(\check{\sigma}). \end{aligned}$$

Furthermore, $\acute{\sigma}_3(\alpha) = \check{\sigma}_3(\alpha)[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}_3(\alpha), \vec{\tau}}]$, and with $\check{\sigma}_3(\alpha) = \check{\sigma}_2(\alpha) = \check{\sigma}(\alpha)$ and

$$\llbracket \vec{E}(z) \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}} = \llbracket \vec{e}[z/\text{this}] \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}} = \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}(\alpha), \vec{\tau}} = \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}_3(\alpha), \vec{\tau}},$$

we get $\acute{\sigma}_3(\alpha) = \acute{\sigma}(\alpha)$. For the new object, $\acute{\sigma}_3(\beta) = \check{\sigma}_3(\beta) = \sigma_{inst}^{init}[\text{this} \mapsto \beta] = \check{\sigma}(\beta) = \acute{\sigma}(\beta)$. Finally, for all other objects γ different from both α and β from the domain of $\acute{\sigma}$ we have $\acute{\sigma}_3(\gamma)(\mathbf{h}_{comm}) = \check{\sigma}_3(\gamma)(\mathbf{h}_{comm}) = \check{\sigma}_1(\gamma)(\mathbf{h}_{comm}) = \acute{\sigma}(\gamma)(\mathbf{h}_{comm})$.

Similarly for the postcondition p_3 of the observation,

$$\begin{aligned} \llbracket P_3(z)[\vec{E}(z)/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}} &= \llbracket P_3(z) \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} \\ &= \llbracket p_3[z/\text{this}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} = \llbracket p_3 \rrbracket_{\mathcal{L}}^{\acute{\omega}, \acute{\sigma}(\alpha), \vec{\tau}} = \llbracket p_3 \rrbracket_{\mathcal{L}}^{\acute{\omega}, \acute{\sigma}_3(\alpha), \vec{\tau}}. \end{aligned}$$

Thus we have to show the existence of a reachable configuration with a global state defining the same instance state for α as $\acute{\sigma}_3$ and containing the local configuration $(\alpha, \vec{\tau}, stm)$. The configuration $\langle T_3, \acute{\sigma}_3 \rangle$ satisfies the above requirements. \square

Lemma C.8. (Cooperation test: Exception handling)

The proof outline $prog'$ satisfies the verification conditions of the cooperation test for exception handling of Definition 6.7.

Proof of Lemma C.8: The proof is analogous to the proof for the cooperation test for communication. Let $u_{ret} := e_0.m(\vec{e}) \langle stm \rangle^{call} \{p_1\}^{wait} \{p_2\}^{?ret} \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} \{p_3\}$ be a statement in a class c with $m \neq \text{start}$ and e_0 of type c' , and let $\{q_1\} \text{ throw } e \{q_2\}^{throw} \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw}$ be a statement in $m(\vec{u})$ of c' which is not in the try-block of any try-catch-finally statement. We have to show that

$$\begin{aligned} \dot{\omega}, \check{\sigma} \models_{\mathcal{G}} & GI \wedge P_1(z) \wedge Q'_1(z') \wedge \text{comm} \\ & \rightarrow (P_2(z) \wedge Q'_2(z')) \circ f_{throw} \wedge (GI \wedge P_3(z)) \circ f_{obs2} \circ f_{obs1} \circ f_{throw} \end{aligned}$$

holds for arbitrary $\dot{\omega}$ and $\check{\sigma}$, with distinct fresh logical variables $z \in LVar^c$ and $z' \in LVar^{c'}$, and with comm given by $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge E'(z') \neq \text{null} \wedge z \neq \text{null} \wedge z' \neq \text{null}$. Furthermore, f_{throw} is $[E'(z')/\text{top}]$, f_{obs1} is $[\vec{E}'_3(z')/z'.\vec{y}_3]$, and f_{obs2} is $[\vec{E}_4(z)/z.\vec{y}_4]$.

So assume that the antecedent holds. From $\dot{\omega}, \check{\sigma} \models_{\mathcal{G}} GI$ we get that there exists a reachable $\langle T, \sigma \rangle$ with

$$\text{dom}(\check{\sigma}) = \text{dom}(\sigma) \text{ and } \check{\sigma}(\gamma)(\mathbf{h}_{comm}) = \sigma(\gamma)(\mathbf{h}_{comm}) \text{ for all } \gamma \in \text{dom}(\sigma).$$

Assuming $\dot{\omega}(z) = \alpha$ as caller identity, $\dot{\omega}, \check{\sigma} \models_{\mathcal{G}} P_1(z)$ implies $\dot{\omega}, \check{\sigma}(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} p_1$ by the substitution Lemma 2.1, for some local state $\hat{\tau}_1$ with $\hat{\tau}_1(u) = \dot{\omega}(u)$ for all local variables u occurring in p_1 . By the annotation definition there exists a reachable configuration $\langle T_1, \sigma_1 \rangle$ such that

$$\sigma_1(\alpha) = \check{\sigma}(\alpha) \text{ and } (\alpha, \hat{\tau}_1, \text{receive } u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1) \in T_1.$$

Recall that $\sigma(\gamma)(\mathbf{h}_{comm}) = \check{\sigma}(\gamma)(\mathbf{h}_{comm})$ for all $\gamma \in \text{dom}(\sigma)$, and especially for the caller $\sigma(\alpha)(\mathbf{h}_{comm}) = \check{\sigma}(\alpha)(\mathbf{h}_{comm}) = \sigma_1(\alpha)(\mathbf{h}_{comm})$. Using the global merging Lemma 7.2 applied to $\langle T_1, \sigma_1 \rangle$ and $\langle T, \sigma \rangle$ we get that there is a reachable $\langle T', \sigma' \rangle$ with $\text{dom}(\sigma') = \text{dom}(\sigma)$ and

$$\sigma'(\alpha) = \sigma_1(\alpha) \text{ and } \sigma'(\gamma) = \sigma(\gamma) \text{ for all } \gamma \in \text{dom}(\sigma) \setminus \{\alpha\}.$$

Furthermore, $(\alpha, \hat{\tau}_1, \text{receive } u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1) \in T_1$, $\sigma_1(\alpha) = \sigma'(\alpha)$, and the local merging Lemma 7.1 implies that

$$(\alpha, \hat{\tau}_1, \text{receive } u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1) \in T' .$$

Let $\beta = \hat{\omega}(z')$ be the callee object. The assumption $\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} Q'_1(z')$ implies $\hat{\omega}, \hat{\sigma}(\beta), \hat{\tau}_2 \models_{\mathcal{L}} q_1$ with $\hat{\tau}_2(v) = \hat{\omega}(v')$ for all local variables v in q_1 . By definition of q_1 there is a reachable global configuration $\langle T_2, \sigma_2 \rangle$ such that

$$\sigma_2(\beta) = \hat{\sigma}(\beta) \text{ and } (\beta, \hat{\tau}_2, \text{throw } e; \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} stm_2) \in T_2 .$$

In case of a self-call, i.e., for $\alpha = \beta$, we directly get that $\langle T'', \sigma'' \rangle = \langle T', \sigma' \rangle$ is a reachable configuration such that $\sigma''(\alpha) = \hat{\sigma}(\alpha) = \hat{\sigma}(\beta)$, $\sigma''(\gamma)(h_{comm}) = \hat{\sigma}(\gamma)(h_{comm})$ for all $\gamma \in dom(\hat{\sigma})$, and $(\alpha, \hat{\tau}_1, \text{receive } u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1) \in T''$. With the local merging lemma we get additionally $(\beta, \hat{\tau}_2, \text{throw } e; \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} stm_2) \in T''$.

Otherwise, for a non-self-call, we need to fall back upon the two merging lemmas once more to obtain a common reachable configuration: Analogously to the caller part, the global merging Lemma 7.2 applied to $\langle T_2, \sigma_2 \rangle$ and $\langle T', \sigma' \rangle$ yields that there is a reachable configuration $\langle T'', \sigma'' \rangle$ with $dom(\sigma'') = dom(\sigma')$ and

$$\sigma''(\beta) = \sigma_2(\beta) \text{ and } \sigma''(\gamma) = \sigma'(\gamma) \text{ for all } \gamma \in dom(\sigma') \setminus \{\beta\} .$$

Now, $(\alpha, \hat{\tau}_1, \text{receive } u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1) \in T'$, $\sigma''(\alpha) = \sigma'(\alpha)$, and the local merging Lemma 7.1 implies that the local configuration $(\alpha, \hat{\tau}_1, \text{receive } u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1)$ is in T'' . Similarly, $(\beta, \hat{\tau}_2, \text{throw } e; \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} stm_2) \in T_2$, $\sigma''(\beta) = \sigma_2(\beta)$, and the local merging Lemma 7.1 implies $(\beta, \hat{\tau}_2, \text{throw } e; \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} stm_2) \in T''$.

Thus $\langle T'', \sigma'' \rangle$ is a reachable configuration with $\sigma''(\alpha) = \hat{\sigma}(\alpha)$, $\sigma''(\beta) = \hat{\sigma}(\beta)$, $\sigma''(\gamma)(h_{comm}) = \hat{\sigma}(\gamma)(h_{comm})$ for all $\gamma \in dom(\hat{\sigma})$, $(\alpha, \hat{\tau}_1, \text{receive } u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1) \in T''$ and $(\beta, \hat{\tau}_2, \text{throw } e; \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} stm_2) \in T''$.

With the antecedent $\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} \text{comm}$ of the cooperation test we get $\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge E'(z') \neq \text{null} \wedge z \neq \text{null} \wedge z' \neq \text{null}$, and by the lifting substitution lemma $\llbracket e_0 \rrbracket_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1} = \llbracket e_0 \rrbracket_{\mathcal{E}}^{\sigma''(\alpha), \hat{\tau}_1} = \hat{\omega}(z') = \beta$. Furthermore, using the same lemma gives $\llbracket \vec{u} \rrbracket_{\mathcal{E}}^{\hat{\omega}, \sigma''(\beta), \hat{\tau}_2} = \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\hat{\omega}, \sigma''(\alpha), \hat{\tau}_1}$ and $\llbracket e \rrbracket_{\mathcal{E}}^{\sigma''(\beta), \hat{\tau}_2} \neq \text{null}$. I.e., the values of the formal and actual parameters agree, and thus the augmentation definition and Lemma B.2 assures that the local configurations are in caller-callee relationship. Additionally, the value of the exception to be thrown is not the empty reference, and thus the exception throwing is enabled.

The definition of the augmentation, and $\sigma''(\alpha) = \hat{\sigma}(\alpha)$ gives

$$\hat{\omega}, \hat{\sigma}(\alpha), \hat{\tau}_1 \models_{\mathcal{L}} p_2 ,$$

with $\hat{\tau}_1 = \hat{\tau}_1[\text{top} \mapsto \llbracket e \rrbracket_{\mathcal{E}}^{\sigma''(\beta), \hat{\tau}_2}]$, which by the substitution Lemma 2.1 and with the definition of $\hat{\tau}_1$ implies that $\hat{\omega}[\text{top} \mapsto \llbracket e \rrbracket_{\mathcal{E}}^{\sigma''(\beta), \hat{\tau}_2}], \hat{\sigma} \models_{\mathcal{G}} P_2(z)$, i.e.,

$$\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} P_2(z) \circ f_{comm} .$$

Since the local state of the callee is not modified during exception throwing, the annotation definition implies $\hat{\omega}, \hat{\sigma}(\beta), \hat{\tau}_2 \models_{\mathcal{L}} q_2$, i.e., $\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} Q'_2(z')$. Due to the renaming mechanism we get

$$\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} Q'_2(z') \circ f_{comm} .$$

Directly after communication we have a global configuration with still the same global state σ'' . The callee observation evolves the global state to $\check{\sigma} = \sigma''[\beta.\vec{y}_3 \mapsto \llbracket \vec{e}_3 \rrbracket_{\mathcal{E}}^{\sigma''(\beta), \check{\tau}_2}]$. Finally, the caller observation changes the global state to $\acute{\sigma} = \check{\sigma}[\alpha.\vec{y}_4 \mapsto \llbracket \vec{e}_4 \rrbracket_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$, where its own local state is updated to $\acute{\tau}_1 = \check{\tau}_1[\vec{y}_4 \mapsto \llbracket \vec{e}_4 \rrbracket_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$. According to the annotation definition we get

$$\hat{\omega}, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3 \quad \text{and} \quad \hat{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI.$$

Let $\acute{\omega} = \hat{\omega}[\text{top} \mapsto \llbracket e \rrbracket_{\mathcal{E}}^{\acute{\sigma}(\alpha), \acute{\tau}_1}][\vec{y}_3 \mapsto \llbracket \vec{e}_3 \rrbracket_{\mathcal{E}}^{\acute{\sigma}(\beta), \acute{\tau}_2}][\vec{y}_4 \mapsto \llbracket \vec{e}_4 \rrbracket_{\mathcal{E}}^{\acute{\sigma}(\alpha), \acute{\tau}_1}]$. The lifting lemma implies $\acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI \wedge P_3(z)$; with the global substitution lemma finally

$$\hat{\omega}, \acute{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z))[\vec{E}_4(z)/z.\vec{y}_4][\vec{E}'_3/z'.\vec{y}_3][E'(z')/\text{top}],$$

and thus the cooperation test for exception handling is satisfied for this case. The case for rethrowing is analogous. \square

Proof of Theorem 7.2 on page 432: Straightforward using the Lemmas C.2, C.3, C.4, C.5, C.6, and C.8, and C.7. \square

Copyright of *Fundamenta Informaticae* is the property of IOS Press and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.

Copyright of *Fundamenta Informaticae* is the property of IOS Press and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.

Copyright of *Fundamenta Informaticae* is the property of IOS Press and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.