Inductive Proof Outlines for Exceptions in Multithreaded Java [1]

Erika Ábrahám [2]

Albert-Ludwigs-University Freiburg, Germany

Frank S. de Boer [4]

CWI Amsterdam, The Netherlands

Willem-Paul de Roever and Martin Steffen [3]

Institute for Computer Science and Applied Mathematics, Christian-Albrechts-Universität zu Kiel, Germany

Abstract

In this paper we give an operational semantics and introduce an assertional proof system for exceptions in a multithreaded *Java* sublanguage.

Key words: *Java*, multi-threading, exceptions, proof systems

1 Introduction

In this work we present an assertional proof system for a multithreaded sublanguage of *Java*, including object and thread creation, aliasing, method call, recursion, *Java*'s synchronization mechanism, and especially exception handling, but ignoring the issues of inheritance and subtyping. The proof system is sound and relatively complete, and allows also to prove deadlock freedom [3].

Verification proceeds in three phases: First the program is *augmented* by fresh auxiliary variables and *annotated* with assertions in the style of Floyd [8,9] intended to hold during program execution when the control flow reaches

[1] Part of this work has been financially supported by IST project Omega (IST-2001-33522), see `http://www-omega.imag.fr`, and NWO/DFG project Mobi-J (RO 1122/9-{1,2,4}.

[2] `mailto:eab@informatik.uni-freiburg.de`

[3] `mailto:{wpr,ms}@informatik.uni-kiel.de`

[4] `mailto:frb@cwi.nl`

the annotated point. Afterwards, the proof system, applied to the augmented and annotated program, also called proof outline [18], yields a number of *verification conditions* which assure that each program execution conforms to the annotation. Finally, the verification conditions must be *proven*. We use the theorem prover *PVS* [19] for this purpose.

With augmentation and annotation provided by the user, the *Verger tool* takes care of verification condition generation in the second phase. The actual verification within the theorem prover is interactive. For the examples we did most of the conditions could be discharged automatically using the built-in proof strategies of *PVS*. Human interaction was needed mostly for the proof of properties whose formulation required quantifiers.

To support a clean interface between internal and external object behavior, we *exclude 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. To mirror this modularity of the program behavior, the assertion logic and the proof system are formulated at two levels, a *local* one for internal object behavior, and a *global* one to describe the global behavior, including the communication topology of objects.

Correctness of the program is guaranteed by the verification conditions of the proof system. These are grouped —besides initial correctness— as follows. The execution of a single method body in isolation is captured by *local correctness* conditions, using the local language. Interference between concurrent method executions is covered by the *interference freedom test* [13,18], formulated also in the local language. It especially has to accommodate reentrant code and the synchronization mechanism. Possibly affecting more than one object, communication and object creation is covered by the *cooperation test*, using the global language. 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 [5] and in [13] for CSP.

This work extends earlier results [2] by exception handling. To our knowledge this is the first sound and relatively complete assertional proof method for a concurrent *Java* sublanguage including synchronization and exception handling. Research in the field of verification for object-oriented programs mostly focused on sequential languages. In particular, there are several works dealing with exceptions and the corresponding proof theory. However, while the concepts of concurrency and exceptions can be understood independently, the combination of both, requires a careful examination of the interaction: the proof system must additionally accomodate the interleaving aspects during exception handling. In particular, we need refined specifications (augmentation and annotation) allowing to describe all interleaving points during exception

handling. Furthermore, besides local correctness for the control structure of exceptions, we need to extend the verification conditions to cover interference during exception handling, and communication due to exceptional return. While the presentation of this paper concentrates on exceptions, the rules are also representative for the whole proof system concerning concurrency.

The LOOP-project [10,14], for instance, develops methods and tools for the verification of sequential object-oriented languages using *PVS* and *Isabelle/HOL*. Especially [12,11] formalize the exception mechanism of *Java*. Poetzsch-Heffter and Müller [21] present a Hoare-style programming logic for a sequential kernel of *Java*. Translating the operational and the axiomatic semantics into the HOL theorem prover allows a computer-assisted soundness proof. In [22] a large subset of *JavaCard*, including exception handling, is formalized in *Isabelle/HOL*, and its soundness and completeness is shown within the theorem prover. In [16] an executable formalization of a simplified version of JVM within the theorem prover ACL2 is given. Early formal accounts of exception mechanisms have been given e.g., in [6] or [15].

The rest of the paper is organized as follows: After introducing the programming language and the assertion language in Sections 2 and 3, the main Section 4 defines the proof system. Section 5 contains some concluding remarks.

2 The programming language

2.1 Syntax

Though we use the abstract syntax for the theoretical part of this work (see Figure 1), our tool supports *Java* syntax. The language is strongly typed; besides class types c, we use booleans and integers as primitive types, and pairs and lists as composite types. We omit the types when this causes no confusion.

Programs are collections of *classes* containing method declarations. We syntactically distinguish between *instance variables* x of classes and *local variables* u of methods; y denotes arbitrary variables. We omit syntactical variable declarations in the abstract syntax, and assume that each variable is appropriately typed. The set of instance variables of a class is implicitly given by the instance variables occurring in that class, and similarly for local variables of methods.

Note that we do not allow qualified references $e.x$ in expressions e. The syntax includes statements for the usual control constructs, especially for throwing and handling exceptions. We use $body_{m,c}$ to denote the body of method m of class c, which must be terminated by a single return statement. Methods can be declared as *non-synchronized* or *synchronized*, using the mod-

$$
\begin{aligned}
e &::= x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(e,\ldots,e) \\
e_{ret} &::= \epsilon \mid e \\
stm &::= x := e \mid u := e \mid u := \mathsf{new}^c \mid u := e.m(e,\ldots,e) \mid e.m(e,\ldots,e) \\
&\mid \mathsf{throw}\ e \mid \mathsf{try}\ stm\ \mathsf{catch}\ (c\,u)\ stm \ldots \mathsf{catch}\ (c\,u)\ stm\ \mathsf{finally}\ stm\ \mathsf{yrt} \\
&\mid \epsilon \mid stm; stm \mid \mathsf{if}\ e\ \mathsf{then}\ stm\ \mathsf{else}\ stm\ \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ stm\ \mathsf{od} \ldots \\
modif &::= \mathsf{nsync} \mid \mathsf{sync} \\
meth &::= modif\, m(u,\ldots,u)\{\ stm; \mathsf{return}\ e_{ret}\} \\
meth_{\mathsf{run}} &::= \mathsf{nsync}\ \mathsf{run}()\{\ stm; \mathsf{return}\ \} \\
meth_{predef} &::= meth_{\mathsf{start}}\ meth_{\mathsf{wait}}\ meth_{\mathsf{notify}}\ meth_{\mathsf{notifyAll}} \\
class &::= \mathsf{class}\ c\{meth \ldots meth\ meth_{\mathsf{run}}\ meth_{predef}\} \\
class_{\mathsf{main}} &::= class \\
prog &::= \langle class \ldots class\ class_{\mathsf{main}}\rangle
\end{aligned}
$$

Fig. 1. Abstract syntax of the programming language

ifiers nsync and sync, respectively.[5] Each class contains the predefined start method, the predefined monitor methods wait, notify, and notifyAll, and a user-defined method run. For the syntactical definition of the predefined methods see [2]. The run method of the main class specifies the entry point of the program.

We require that the local variable u in blocks catch $(c\,u)$ *stm* does not occur in the same method outside the catch block. Furthermore, object creation and method call statements may not contain instance variables, and formal parameters may not be assigned to.[6] The run methods cannot be invoked directly.

2.2 Semantics

2.2.1 States and configurations

In the semantics we add the type Object as the supertype of all classes. Note that no objects of type Object can be created, thus preserving monomorphism. Let Val^t be the disjoint domains of the various types t. For class names c, we use $\alpha, \beta, \ldots \in Val^c$ as typical elements for *object identifiers*. The value of null

[5] *Java* does not have a non-synchronized modifier; methods are declared non-synchronized by default.

[6] These restrictions could be relaxed but it would increase the complexity of the proof system.

in type c is $null^c \notin Val^c$.

A *local state* $\tau_{m,c}$ (or short τ) of method m of class c holds the values of the method's local variables. The initial local state is denoted by τ^{init}. A *local configuration* (α, τ, stm) of a method of an object $\alpha \neq null$ specifies its local state τ and its control point represented by the statement stm. A *thread configuration* ξ is a stack $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n)$ representing the call chain of a thread. We write $\xi \circ (\alpha, \tau, stm)$ for pushing a new local configuration onto stack ξ.

An object is characterized by its *instance state* σ_{inst}, assigning values to the self-reference and to the instance variables. The initial instance state is denoted by σ_{inst}^{init}. A *global state* σ stores for each currently *existing* object α its instance state $\sigma(\alpha)$ with the invariant property $\sigma(\alpha)(\mathsf{this}) = \alpha$. A *global configuration* $\langle T, \sigma \rangle$ describes the currently existing objects by the global state σ, where the set T contains the configurations of all currently executing threads. Expressions are evaluated in the context of an instance state and a local state with main cases $[\![x]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \sigma_{inst}(x)$ and $[\![u]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \tau(u)$.

The local state $\tau[u \mapsto v]$ results from τ by assigning the value v to the variable u; $\sigma_{inst}[x \mapsto v]$ is similar, and $\sigma[\alpha.x \mapsto v]$ results from σ by assigning v to the instance variable x of the object α. We also use $\tau[\boldsymbol{y} \mapsto \boldsymbol{v}]$, where instance variables occurring in $\boldsymbol{y}$ are untouched; $\sigma_{inst}[\boldsymbol{y} \mapsto \boldsymbol{v}]$ and $\sigma[\alpha.\boldsymbol{y} \mapsto \boldsymbol{v}]$ are similar, where instance variables are untouched. For global states, $\sigma[\alpha \mapsto \sigma_{inst}]$ equals σ except on α; note that if α is not in the domain $dom(\sigma)$ of σ, the operation extends the set of existing objects by α with initial state σ_{inst}.

2.2.2 Operational semantics

In this section we present the operational semantics of exception handling (cf. Figure 3). Additionally, we list in Figure 2 some rules for the normal flow of

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

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

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

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

Fig. 2. Operational semantics (1)

control; for the remaining transition rules we refer to [2].

$$\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{try}\ stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle \qquad \textsc{Try}$$

$$\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{yrt}; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle \qquad \textsc{Finally}_{out}$$

$$\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{yrt}_\beta; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{throw}\ \beta; stm)\}, \sigma\rangle \qquad \textsc{Finally}^{exc}_{out}$$

$$\frac{n \geq 0}{\begin{array}{l}\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{catch}\ (c_1\ u_1)\ stm_1 \ldots \mathsf{catch}\ (c_n\ u_n)\ stm_n\ \mathsf{finally}\ stm\ \mathsf{yrt}; stm')\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm\ \mathsf{yrt}; stm')\}, \sigma\rangle\end{array}} \textsc{Finally}_{in}$$

$$\frac{\begin{array}{c}stm \text{ is try-closed} \qquad stm' = \mathsf{catch}\ (c_1\ u_1)\ stm_1 \ldots \mathsf{catch}\ (c_n\ u_n)\ stm_n\ \mathsf{finally}\ stm_{n+1}\ \mathsf{yrt} \\ 1 \leq i \leq n \qquad [\![e]\!]^{\sigma(\alpha),\tau}_{\mathcal{E}} \in Val^{c_i} \qquad \forall 1 \leq j < i.\ [\![e]\!]^{\sigma(\alpha),\tau}_{\mathcal{E}} \notin Val^{c_j} \\ \tau' = \tau[u_i \mapsto [\![e]\!]^{\sigma(\alpha),\tau}_{\mathcal{E}}]\end{array}}{\begin{array}{l}\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{throw}\ e; stm\, stm'; stm'')\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau', stm_i\ \mathsf{finally}\ stm_{n+1}\ \mathsf{yrt}; stm'')\}, \sigma\rangle\end{array}} \textsc{Catch}$$

$$\frac{\begin{array}{c}stm \text{ is try-closed} \qquad stm' = \mathsf{catch}\ (c_1\ u_1)\ stm_1 \ldots \mathsf{catch}\ (c_n\ u_n)\ stm_n\ \mathsf{finally}\ stm_{n+1}\ \mathsf{yrt} \\ [\![e]\!]^{\sigma(\alpha),\tau}_{\mathcal{E}} = \beta \neq null \qquad 0 \leq n \qquad \forall 1 \leq i \leq n.\ [\![e]\!]^{\sigma(\alpha),\tau}_{\mathcal{E}} \notin Val^{c_i}\end{array}}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{throw}\ e; stm\, stm'; stm'')\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm_{n+1}\mathsf{yrt}_\beta; stm'')\}, \sigma\rangle} \textsc{Finally}^{exc}_{in}$$

$$\frac{stm \text{ is try-closed} \qquad [\![e]\!]^{\sigma(\alpha),\tau}_{\mathcal{E}} = \beta \neq null}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{throw}\ e; stm\ \ \mathsf{yrt}_{\beta'}; stm')\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{yrt}_\beta; stm')\}, \sigma\rangle} \textsc{ThrowFinally}$$

$$\frac{stm' \text{ is try-closed} \qquad [\![e]\!]^{\sigma(\beta),\tau'}_{\mathcal{E}} = \gamma \neq null}{\begin{array}{l}\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{receive}\ u_{ret}; stm) \circ (\beta, \tau', \mathsf{throw}\ e; stm')\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{throw}\ \gamma; stm)\}, \sigma\rangle\end{array}} \textsc{Return}^{exc}$$

$$\frac{stm \text{ is try-closed} \qquad [\![e]\!]^{\sigma(\alpha),\tau}_{\mathcal{E}} = \beta \neq null}{\langle T \dot\cup \{(\alpha, \tau, \mathsf{throw}\ e; stm; \mathsf{return})\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{(\alpha, \tau, \mathsf{return}_\beta)\}, \sigma\rangle} \textsc{Terminate}^{exc}$$

Fig. 3. Operational semantics (2)

The *initial configuration* $\langle T_0, \sigma_0\rangle$ of a program satisfies $dom(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma^{init}_{inst}[\mathsf{this} \mapsto \alpha]$, and $T_0 = \{(\alpha, \tau^{init}, body_{\mathsf{run},c})\}$, 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 exists a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle$ such that $\langle T_0, \sigma_0\rangle$ is an initial configuration and $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$.

Exceptions allow a special form of error handling: If something unexpected or unallowed happens, the executing thread may throw an exception

object.[7] A thrown exception interrupts the normal control flow which moves to the "nearest" exception handler handling exceptions of the given type, as explained below.

Generally, a try-catch-finally block is executed from left to right. An exception raised in the try part transfers control to the first matching catch-clause, if any, which may raise an exception of its own. The finally part is executed independently of whether an exception is raised at all or whether it has been caught or "fallen through". If the finally-clause throws an exception, it *overrides* the uncaught exception of previous parts, if any.

During the execution of a try-catch-finally block $\mathsf{try}\ stm_0\ \mathsf{catch} \ldots \mathsf{yrt}$, the executing local configuration contains an "open" block like, e.g., $stm_0'\ \mathsf{catch} \ldots \mathsf{yrt}$. We also call such blocks statements, even if they are not statements in the original syntax. Statements in which no such open blocks occur are called *try-closed*.

After entering a try-catch-finally statement (cf. rule Try) the try clause is executed until it terminates or an exception is thrown. A thrown exception is caught by the first catch clause handling exceptions of the given type, if any (cf. rule Catch); afterwards, execution continues with the finally clause (Finally$_{in}$). If uncaught (cf. rule Finally$^{exc}_{in}$), control moves directly into the finally clause. The local configuration remembers the uncaught exception β by the notation yrt_β. The same rule takes care of exceptions thrown in some catch-clause. Throwing an exception in the finally-clause replaces uncaught exceptions thrown in the try- or catch-clauses (cf. rule ThrowFinally). An analogous rule where β' is not present covers the case without an uncaught exception. With no exceptions thrown in the try clause, only the finally clause gets executed (cf. rule Finally$_{in}$).

With no pending exception at the end (cf. rule Finally$_{out}$), the finally-clause continues with the normal control flow. Otherwise, the uncaught exception β gets rethrown (cf. rule Finally$^{exc}_{out}$) using the auxiliary statement $\mathsf{throw}\,\beta$. For all rules defining the semantics of throw statements $\mathsf{throw}\,e$ we have an analogous rule for rethrowing $\mathsf{throw}\,\alpha$. Throwing an exception outside try-catch-finally blocks causes the control to return to the caller, and to rethrow the exception there. As threads start their execution in run methods, throwing an exception outside try-catch-finally blocks in run methods terminates the executing thread abnormally (cf. rules Return^{exc} and Terminate^{exc}).

[7] In contrast to *Java*, in our language all objects may serve as exceptions.

3 The assertion language

The assertion logic consists of a *local* and a *global* sublanguage. *Local assertions* $p, q, \ldots$ are used to annotate methods in terms of their local variables and of the instance variables of the class to which they belong, and are evaluated in the context of some states $(\omega, \sigma_{inst}, \tau)$. *Global assertions* $P, Q, \ldots$, evaluated in the context of some (ω, σ), describe a whole system of objects and their communication structure and will be used in the cooperation test.

Assertions are built using the usual predicate logic constructs. *Logical variables* $z \in LVar$, different from all program variables, are used for quantification and as free variables to represent local variables in the global language. Logical environments ω assign values to logical variables. By $\mathsf{hastype}(e, c)$ we state that the value of e is of type c; we use this construct to express the type of exceptions. Since the programming language is monomorphically typed, the association is unique. Qualified references $E.x$ may be used in the global language only.

For class types, quantification in the global language ranges over the set of *existing* objects and *null*. In contrast, one can quantify over objects on the local level only if the domain of quantification is explicit: $\exists z{\in}e.p$ states that there is a value in the sequence e, for which p holds; $\exists z{\sqsubseteq}e.p$ states the existence of a subsequence.

$$
\begin{aligned}
e &::= z \mid x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(e, \ldots, e) \\
p &::= e \mid \neg p \mid p \wedge p \mid \mathsf{hastype}(e, c) \mid \exists z.\ p \mid \exists z \in e.\ p \mid \exists z \sqsubseteq e.\ p \\
\\
E &::= z \mid \mathsf{null} \mid \mathsf{f}(E, \ldots, E) \mid E.x \\
P &::= E \mid \neg P \mid P \wedge P \mid \mathsf{hastype}(E, c) \mid \exists z.\ P
\end{aligned}
$$

We write $[\![\_]\!]^{\_,\_,\_}_{\mathcal{L}}$ and $[\![\_]\!]^{\_,\_}_{\mathcal{G}}$ for the semantic functions evaluating local and global assertions, $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ for $[\![p]\!]^{\omega,\sigma_{inst},\tau}_{\mathcal{L}} = \mathit{true}$, and $\models_{\mathcal{L}} p$ if p holds in all contexts; we use analogously $\models_{\mathcal{G}}$ for global assertions.

To express a local property in the global language, we define the lifting substitution $p[z/\mathsf{this}]$ by simultaneously replacing in p all occurrences of this by z, and all occurrences of instance variables x by qualified references $z.x$, where z is assumed not to occur in p. For notational convenience we view the local variables occurring in $p[z/\mathsf{this}]$ as logical variables. Formally, these local variables are replaced by fresh logical variables. We write $P(z)$ for $p[z/\mathsf{this}]$.

4 The proof system

Next we introduce proof outlines and present the proof method. We formulate verification conditions as Hoare triples. The formal semantics is given in [3] by means of a weakest precondition calculus [7].

4.1 Proof outlines

For a complete proof system the transition semantics must be expressible in the assertion language. As the assertion language reasons about the local and global states, we have to *augment* the program with assignments to *fresh auxiliary variables* which we call *observations* to represent information about the control points and stack structures within the local and global states. Auxiliary variables are variables not occurring in the program and may be added to the program to observe certain aspects of the flow of control without affecting it. In general, the observations record information about the intra-object interleaving of threads, which gives rise to shared-variable concurrency, or the inter-object communication via method calls and returns. As exception throwing is side-effect free, it does not affect the interleaving of threads but only the communication behavior between objects: Exception throwing inside try-catch-finally blocks influences the control flow, but without modifying the instance states. As such, those statements do not need to be observed. Exceptions escaping the method body, i.e., thrown outside any try-catch-finally block, cause the control to return to the caller. I.e., the execution of such statements affect the communication structure and must be observed.

Invariant program properties are specified by an *annotation,* associating assertions with the control points of the augmented program. In contrast to the pre- and post-specifications for methods in a sequential context, in our concurrent setting *each* control point needs to be annotated to capture the effect of interleaving. An augmented and annotated program is called a *proof outline* or an *asserted program.*

4.1.1 Augmentation

Syntactically, an augmentation extends a program by atomically executed multiple assignments $\boldsymbol{y} := \boldsymbol{e}$ to distinct auxiliary variables. Furthermore, the observations have, in general, to be "attached" to statements they observe in an atomic manner. For assignments and object creation, this is represented by the augmentations $y := e \ \langle \boldsymbol{y} := \boldsymbol{e} \rangle^{ass}$ and $u := \mathsf{new}^c \ \langle \boldsymbol{y} := \boldsymbol{e} \rangle^{new}$. Augmentations of exception throwing outside try-catch-finally blocks respectively of top-level try-catch-finally blocks, i.e., those not nested inside further such blocks, are of the form

$$\mathsf{throw}\ e \ \langle \boldsymbol{y}_1 := \boldsymbol{e}_1 \rangle^{throw} \quad \text{resp.} \quad \mathsf{try} \ldots \mathsf{yrt} \ \langle \boldsymbol{y}_2 := \boldsymbol{e}_2 \rangle^{rethrow} .$$

The second augmentation $\boldsymbol{y}_2 := \boldsymbol{e}_2$ observes the rethrowing of exceptions which are thrown but not caught in the block, if any. Nested blocks are not augmented by such an observation. In the augmentation

$$u := e_0.m(\boldsymbol{e})\ \langle \boldsymbol{y}_1 := \boldsymbol{e}_1 \rangle^{!call}\, \langle \boldsymbol{y}_4 := \boldsymbol{e}_4 \rangle^{?ret}\, \langle \boldsymbol{y} := \boldsymbol{e} \rangle^{rethrow}$$

of a top-level method call, $\boldsymbol{y}_1 := \boldsymbol{e}_1$ observes the call, $\boldsymbol{y}_4 := \boldsymbol{e}_4$ observes both normal and exceptional returning, and $\boldsymbol{y} := \boldsymbol{e}$ the rethrowing of an exception in case control returns due to an exception. [8] Augmentations of method calls inside try-catch-finally blocks do not observe rethrowing. For method bodies, the augmentation

$$\langle \boldsymbol{y}_2 := \boldsymbol{e}_2 \rangle^{?call}\ stm; \mathsf{return}\ e_{ret}\ \langle \boldsymbol{y}_3 := \boldsymbol{e}_3 \rangle^{!ret}$$

observes by $\boldsymbol{y}_2 := \boldsymbol{e}_2$ the reception of the call and $\boldsymbol{y}_3 := \boldsymbol{e}_3$ observes normal returning. Abnormal return is observed by the augmentation of top-level exception (re)throwing in the method body. A stand-alone observation $\langle \boldsymbol{y} := \boldsymbol{e} \rangle$ can be inserted at any point in the program.

For assignment, object creation, exception throwing, and exceptions handling, first the statement and then its observation are executed in a single computation step. For communication, where two observations can be involved, the observation of the sender precedes the receiver observation. In the following we call assignments together with their observations also multiple assignments. 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*.

Besides user-definable auxiliary variables, our proof system is formulated in terms of *built-in* auxiliary variables, automatically included into all augmentations. These auxiliary variables record information needed to identify threads and local configurations, and to describe monitor synchronization. In general, updates of the built-in auxiliary variables are described explicitly in the augmentations or implicitly in the verification conditions. The latter in particular is needed to reason about inter-object communication, where the update cannot be determined locally; such an example is given in the cooperation test for abnormal return.

For exception handling, one unique local variable associated with each finally clause stores the exception α associated with the yrt_α in the semantics. Mimicking the operational semantics, entering a finally-clause sets the associated local variable to the last uncaught exception, if any, and to *null* otherwise. Furthermore, we introduce for each method an additional local variable exc to store the value of the exception to be rethrown.

[8] We syntactically exclude the simultaneous execution of two observations in a single computation step in the same object [2].

4.1.2 Annotation

We use the Hoare-triple notation $\{p\}\ stm\ \{q\}$ and write $pre(stm)$ and $post(stm)$ to refer to the pre- and the post-condition of a statement. The annotation

$$\{p_0\}\ \ \mathsf{throw}\ e\ \{p_1\}^{throw}\ \langle \boldsymbol{y} := \boldsymbol{e}\rangle^{throw}\ \ \{p_2\}$$

of exception throwing at the top-level specifies p_0 and p_2 as pre- and postconditions, whereas p_1 at the auxiliary point should hold directly after throwing but before its observation; the case for object creation is similar. Note that the control point at p_2 is not reachable. Non-top-level throw-statements are not observed and therefore do not involve a corresponding annotation p_1. In the annotation

$$\begin{aligned}\{p_0\}\ \ u := e_0.m(\boldsymbol{e})\ \{p_1\}^{!call}\ \langle \boldsymbol{y}_1 := \boldsymbol{e}_1\rangle^{!call}\ \ \{p_2\}^{wait}\\ \{p_3\}^{?ret}\ \langle \boldsymbol{y}_4 := \boldsymbol{e}_4\rangle^{?ret}\ \ \{p_4\}^{exc}\ \ \{p_5\}^{rethrow}\ \ \langle \boldsymbol{y} := \boldsymbol{e}\rangle^{rethrow}\ \ \{p_6\}\end{aligned}$$

the control points are annotated by p_0 as the precondition of the call, p_2 at the point waiting for return, p_4 describes the state after an exceptional return, and p_6 is the postcondition of the method after normal return. The assertions p_1, p_3, and p_5 at the auxiliary points are the preconditions of the corresponding observations. Non-top-level method calls do not observe rethrowing and therefore do not involve a corresponding annotation p_5. The annotation of method bodies is of the form

$$\{p_1\}^{?call}\langle \boldsymbol{y}_2 := \boldsymbol{e}_2\rangle^{?call}\{p_2\}\ stm; \{p_3\}\ \mathsf{return}\ e_{ret}\{p_4\}^{!ret}\langle \boldsymbol{y}_3 := \boldsymbol{e}_3\rangle^{!ret}\{p_5\}\,,$$

where p_2, p_3, and p_5 annotate the corresponding control points, and p_1 and p_4 are the preconditions of the given observations. The annotation of a top-level try-catch-finally block has the form

$$\{p_0\}\ \ \mathsf{try}\ldots\mathsf{yrt}\quad \{p_1\}^{exc}\{p_2\}^{rethrow}\langle \boldsymbol{y} := \boldsymbol{e}\rangle^{rethrow}\{p_3\}\ \ .$$

The assertion p_0 is the precondition of the try-catch-finally block. If an exception is to be rethrown, the assertion p_1 is required to hold after exiting the whole try-catch-finally block, p_2 must hold after rethrowing and prior to its observation $\boldsymbol{y} := \boldsymbol{e}$. Note that this observation does not have a postcondition, as the control point after the observation is not reachable. If no exception needs to be rethrown, the assertion p_3 should hold after exiting the finally-block. Inner try-catch-finally blocks do not observe rethrowing and therefore do not involve a corresponding annotation p_2.

Besides pre- and postconditions, the annotation defines for each class c a local assertion I_c called *class invariant*, specifying invariant properties of instances of c in terms of its instance variables. 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 due to method calls and returns, object creation, or exception throwing or rethrowing outside try-catch-finally blocks in methods different from run. The annotation must not contain free logical variables. In partially annotated statements, assertions which are not explicitly specified are by definition true.

4.2 Verification Conditions

4.2.1 Local correctness

A proof outline is *locally correct*, if the annotation of a method is invariant under its own sequential execution [4]. Below we define local correctness for exception handling; for the remaining rules with a detailed explanation and for the initial conditions see [3].

Definition 1 (Local correctness: Exception handling) *A proof outline is* locally correct *under exception handling, if for all statements stm of the form*

$$\begin{array}{lllll} \{p\}\ \mathsf{try} & \{p_0\} & stm_0 & \{p_0'\} & \\ \quad \mathsf{catch}(c_1\, u_1) & \{p_1\} & stm_1 & \{p_1'\} & \cdots \\ \quad \mathsf{catch}(c_n\, u_n) & \{p_n\} & stm_n & \{p_n'\} & \\ \quad \mathsf{finally} & \{p_{\mathsf{fin}}\} & stm_{\mathsf{fin}} & \{p_{\mathsf{fin}}'\}\ \mathsf{yrt}\ \{p_{\mathsf{exc}}\}^{exc} & \{p'\}\ , \end{array}$$

where u is the built-in auxiliary local variable associated with the finally clause, and for all $0 \leq i \leq n$,

$$\models_{\mathcal{L}} p \ \rightarrow\ p_0\,, \tag{1}$$

$$\models_{\mathcal{L}} \{p_i'\} u := \mathsf{null} \{p_{\mathsf{fin}}\}\,, \tag{2}$$

$$\models_{\mathcal{L}} p_{\mathsf{fin}}' \wedge u = \mathsf{null} \rightarrow p' \tag{3}$$

$$\models_{\mathcal{L}} \{p_{\mathsf{fin}}' \wedge u \neq \mathsf{null}\}\ \mathsf{exc} := u\ \{p_{\mathsf{exc}}\}\,, \tag{4}$$

and for all statements $\{q_0\}$ throw e in stm_0 which is not in a try-catch-finally block inside stm_0, and for all $1 \leq i \leq n$,

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \mathsf{null} \wedge \mathsf{hastype}(e, c_i) \wedge \forall 1 \leq j < i. \neg\, \mathsf{hastype}(e, c_j)\} \quad u_i := e; \quad \{p_i\}\,, \tag{5}$$

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \mathsf{null} \wedge \forall 1 \leq j \leq n. \neg\, \mathsf{hastype}(e, c_j)\}\ u := e;\ \{p_{\mathsf{fin}}\}\,. \tag{6}$$

For statements $\{q_0\}$ throw e in catch blocks, (6) must hold without the antecedent $\forall 1 \leq j \leq n. \neg\, \mathsf{hastype}(e, c_j)$. For throw *statements in finally blocks, (6) must hold without the above antecedent and with p_{fin} replaced by p_{fin}'. The above conditions must hold also for assertions $\{q_0\}^{exc}$ inside try-catch-finally blocks. In this case e in the conditions is replaced by* exc.

4.2.2 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. Without qualified references to instance variables in the programming language, we only have to deal with invariance under execution within the *same* object. Affecting only local variables, exception throwing, 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. Assertions at auxiliary points, which are not interleaving points, do not have to be shown invariant.

Let q be an assertion at a control point and $\boldsymbol{y} := \boldsymbol{e}$ a multiple assignment in the same class. To distinguish local variables of the different local configurations, we replace all local variables u of the assertion q by fresh ones u', resulting in q'. When does q have to be invariant under the execution of the assignment? If the assertion and the assignment belong to the *same* thread, the only assertions endangered are those at control points waiting for return earlier in the thread's stack. However, invariance of a local configuration under its own execution, and interference with the matching return statement in a self-call need not be considered.

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.

The above situations are formalized by an assertion interleavable, using built-in auxiliary variables for the identification of threads and local configurations.

Definition 2 (Interference freedom) *A proof outline is* interference free, *if for all classes c, multiple assignments* $\boldsymbol{y} := \boldsymbol{e}$ *with precondition* p *in* c*, and assertions* q *at control points in* c *the following holds:*

$$\models_{\mathcal{L}} \{p \wedge I_c\} \quad \boldsymbol{y} := \boldsymbol{e} \quad \{I_c\} \tag{7}$$

$$\models_{\mathcal{L}} \{p \wedge q' \wedge \mathsf{interleavable}(q, \boldsymbol{y} := \boldsymbol{e})\} \quad \boldsymbol{y} := \boldsymbol{e} \quad \{q'\}\,. \tag{8}$$

4.2.3 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 communication and object creation. For the case of returning, the cooperation test defines two sets of verification conditions, one for normal return and the other for abnormal return. We restrict here to the verification conditions for abnormal return.

The cooperation test for exception handling covers the (re)throwing of

exceptions outside 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:

$$\begin{array}{lllll} \text{caller: } u_{ret} := e_0.m(\boldsymbol{e}) \ldots & \{p_1\}^{wait} & \{p_2\}^{?ret} & \langle \boldsymbol{y}_4 := \boldsymbol{e}_4 \rangle^{?ret} & \{p_3\}^{exc} \ldots \\ \text{callee: } \ldots & \{q_1\}\ \mathsf{throw}\ e & \{q_2\}^{throw} & \langle \boldsymbol{y}_3 := \boldsymbol{e}_3 \rangle^{throw} & \ldots \end{array}$$

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 (9) 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 (10) below. 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. To distinguish local variables of caller and callee, we rename those of the callee; the result we denote by primed variables, expressions, and assertions.

That the identity of the thrown exception is stored in the local variable exc of the caller is represented by $\mathsf{exc} := E'(z')$. The callee and the caller observations are represented by the assignments $z'.\boldsymbol{y}'_3 := \boldsymbol{E}'_3(z')$ and $z.\boldsymbol{y}_4 := \boldsymbol{E}_4(z)$, respectively.

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 $e_0.m(\boldsymbol{e})$. 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 $\boldsymbol{u}' = \boldsymbol{E}(z)$ states that the values of the formal and of the actual parameters agree. The built-in augmentation defines an auxiliary formal parameter for each method which stores the "return address", i.e., which identifies the caller local configuration. Using this variable, the assertion $E_0(z) = z' \wedge \boldsymbol{u}' = \boldsymbol{E}(z)$ assures that the local configurations are in caller-callee relationship. Furthermore, $E'(z') \neq \mathsf{null}$ expresses that the exception to be thrown is not the null reference.

Similarly to exception throwing using statements throw e outside try-catch-

finally blocks, the cooperation test defines analogous conditions for rethrowing exceptions outside try-catch-finally blocks: Also in these cases, which can occur after method calls or after non-nested try-catch-finally blocks, control returns to the caller.

Definition 3 (Cooperation test: Exception handling) *A proof outline satisfies the* cooperation test for exception handling*, if for all statements*

$$u_{ret} := e_0.m(\boldsymbol{e})\ \langle stm\rangle^{!call}\ \{p_1\}^{wait}\ \{p_2\}^{?ret}\ \langle \boldsymbol{y}_4 := \boldsymbol{e}_4\rangle^{?ret}\ \{p_3\}^{exc}$$

(or those without receiving a value) in class c with e_0 of type c', and for all statements $\{q_1\}$ $\mathsf{throw}\ e\ \{q_2\}^{throw}\langle \boldsymbol{y}_3 := \boldsymbol{e}_3\rangle^{throw}$ in $m(\boldsymbol{u})$ of c' outside any try-catch-finally statement,

$$\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm}\} \tag{9}$$
$$\mathsf{exc} := E'(z') \quad \{P_2(z) \wedge Q_2'(z')\} \quad \textit{and}$$
$$\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm}\} \tag{10}$$
$$\mathsf{exc} := E'(z'); \quad z'.\boldsymbol{y}_3' := \boldsymbol{E}_3'(z'); \quad z.\boldsymbol{y}_4 := \boldsymbol{E}_4(z) \quad \{GI \wedge P_3(z)\}$$

hold with distinct fresh logical variables z of type c and z' of type c', and with comm *given by $E_0(z) = z' \wedge \boldsymbol{u}' = \boldsymbol{E}(z) \wedge E'(z') \neq \mathsf{null} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}$. The same conditions must hold also for top-level rethrowing, i.e., for annotated fragments of the form $\{q_1\}^{exc}\{q_2\}^{rethrow}\langle \boldsymbol{y}_3 := \boldsymbol{e}_3\rangle^{rethrow}$ under the same requirements, where e in the conditions is replaced by* exc*.*

For exception (re)throwing in run methods Conditions (9) and (10) simplify in that there is no caller; see [3] for the corresponding conditions.

5 Conclusion

In this work we presented an assertional proof system for a *Java* sublanguage including concurrency, synchronization, and exception handling, but no inheritance. The proof system is sound and complete, and allows also to prove deadlock freedom [3]. Computer support is given by the tool *Verger* [1]. The tool takes an augmented and annotated *Java* program, i.e., a 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.

For future work, we plan to extend the programming language by further constructs, like inheritance and subtyping along the lines of [20].

References

[1] Ábrahám, E., "An Assertional Proof System for Multithreaded Java — Theory and Tool Support," Ph.D. thesis, University of Leiden (2004), defended 20.1.2005.

[2] Ábrahám, E., F. S. de Boer, W.-P. de Roever and M. Steffen, *Inductive proof-outlines for monitors in Java*, in: Najm et al. [17], pp. 155–169, a longer version appeared as technical report TR-ST-03-1, April 2003

[3] Ábrahám, E., F. S. de Boer, W.-P. de Roever and M. Steffen, *Inductive proof outlines for multithreaded Java with exceptions*, Technical Report 0313, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel (2003).
URL `http://www.informatik.uni-kiel.de/reports/2003/0313.html`

[4] Apt, K. R., *Ten years of Hoare's logic: A survey – part I*, ACM Transactions on Programming Languages and Systems **3** (1981), pp. 431–483.

[5] Apt, K. R., N. Francez and W.-P. de Roever, *A proof system for communicating sequential processes*, ACM Transactions on Programming Languages and Systems **2** (1980), pp. 359–385.

[6] Christian, F., *Correct and robust programs*, ACM Transactions on Programming Languages and Systems **10** (1984), pp. 163–174.

[7] de Boer, F. S., *A WP-calculus for OO*, in: W. Thomas, editor, *Proceedings of FoSSaCS '99*, Lecture Notes in Computer Science **1578** (1999), pp. 135–156.

[8] Floyd, R. W., *Assigning meanings to programs*, , **19**, 1967, pp. 19–32.

[9] Hoare, C. A. R., *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), pp. 576–580.

[10] Huisman, M., "Java Program Verification in Higher-Order Logic with PVS and Isabelle," Ph.D. thesis, University of Nijmegen (2001).

[11] Huisman, M. and B. Jacobs, *Java program verification via a Hoare logic with abrupt termination*, in: T. Maibaum, editor, *Proceedings of FASE'00*, Lecture Notes in Computer Science **1783** (2000), pp. 284–303.

[12] Jacobs, B., *A formalisation of Java's exception mechanism*, in: D. Sands, editor, *Proceedings of ESOP 2001*, Lecture Notes in Computer Science **2028** (2001), pp. 284–301.

[13] Levin, G. and D. Gries, *A proof technique for communicating sequential processes*, Acta Informatica **15** (1981), pp. 281–302.

[14] *The LOOP project: Formal methods for object-oriented systems*, `http://www.cs.kun.nl/~bart/LOOP/` (2001).
URL `http://www.cs.kun.nl/~bart/LOOP/`

[15] Manasse, M. S. and C. G. Nelson, *Correct compilation of control structures*, Technical memo, Bell Laboratories (1984).

[16] Moore, J. S. and G. M. Porter, *An executable formal Java Virtual Machine thread model*, in: *Proceedings of the 2001 JVM Usenix Symposium in Monterey, California*, 2001.

[17] Najm, E., U. Nestmann and P. Stevens, editors, "Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '03), Paris," Lecture Notes in Computer Science **2884**, Springer-Verlag, 2003.

[18] Owicki, S. and D. Gries, *An axiomatic proof technique for parallel programs*, Acta Informatica **6** (1976), pp. 319–340.

[19] Owre, S., J. M. Rushby and N. Shankar, *PVS: A prototype verification system*, in: D. Kapur, editor, *Automated Deduction (CADE-11)*, Lecture Notes in Computer Science **607** (1992), pp. 748–752.

[20] Pierik, C. and F. S. de Boer, *A syntax-directed Hoare logic for object-oriented programming concepts*, in: Najm et al. [17], pp. 64–78, an extended version appeared as University of Utrecht Technical Report UU-CS-2003-010.

[21] Poetzsch-Heffter, A. and P. Müller, *A programming logic for sequential Java*, in: S. Swierstra, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science **1576** (1999), pp. 162–176.

[22] von Oheimb, D. and T. Nipkow, *Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited*, in: L.-H. Eriksson and P. A. Lindsay, editors, *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)*, Lecture Notes in Computer Science **2391** (2002), pp. 89–105.