Utrecht University
Faculty of Science
Department of Information and Computing Sciences

The Path Explosion Problem in Symbolic Execution

An Approach to the Effects of Concurrency and Aliasing

Author
S. (Stefan) Koppier

Supervisor
dr. S.W.B. (Wishnu) Prasetya

Second Examiner
prof. dr. G.K. (Gabriele) Keller

May 5, 2020

Abstract

Symbolic Execution is a technique used for the formal verification of software. A notorious problem with formal verification is the path explosion problem: the exponentially increasing requirement of computing power to verify more complex software. Common contributors to the path explosion problem are: (1) loops and recursion; (2) exceptions; (3) pointer aliasing; and (4) concurrency. In this thesis, we present an intermediate verification language: the OOX language. OOX is intended as a core language for object-oriented languages with support for concurrency, as Java and C#. We developed a symbolic execution engine for OOX, which aims to limit the effects of the path explosion problem resulting from the first, third and fourth contributors. To achieve this, we applied three main optimizations: partial order reduction, formula caching and expression evaluation. Furthermore, the logic to handle references is defined in the front-end of our approach. We use a SMT solver as a back-end to give a definitive answer when the front-end does not give a definitive answer. The results of this approach show a positive image. The optimizations drastically reduce the runtime and our approach can compete with existing tools like CBMC and CIVL.

Keywords: Formal verification, symbolic execution, software testing, partial order reduction.

Definitions

Acronyms

AST Abstract Syntax Tree. 9, 36, 37, 41

CFA Control Flow Analysis. 39

CFG Control Flow Graph. 36, 39–41, 43, 49, 81

CSL Concurrent Separation Logic. 80

IVL Intermediate Verification Language. 78, 79, 81

POR Partial Order Reduction. 63–66, 74, 82

SE Symbolic Execution. 35, 65

SEE Symbolic Execution Engine. 35–37, 39, 40, 44, 45, 55, 60, 61, 67, 70, 75, 82

SS Symbolic State. 35, 36, 44, 46–49, 51, 65, 66

Contents

Chapter 1

Introduction

The importance in our society is ever increasing. A major aspect of the software engineering field is that the software that is developed works as expected. All software that is developed is verified for correctness to some degree. The IEEE defines verification as "The evaluation of whether or not a system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process, in contrast to validation, which aims to assure that the system meets the needs of the stakeholders" [1].

There exist several approaches to verify the correctness of software, ranging from manual testing to automated exhaustive approaches, such as (symbolic) model checking and symbolic execution. In manual approaches, specific test instances are created to execute the program, whether or not with the help of existing tools, checking that for those specific instances the program is correct. Automated approaches aim to verify that for all possible inputs, all possible executions are correct. In model checking, a transition system is constructed, which represents the program under verification. It is common for the specification to be described in some temporal logic. The transition system is regularly described using Kripke structures. In symbolic model checking, the transition system is modelled with logical formulas, in contrast to an explicit transition system. Symbolic execution is a technique where the program under verification is executed in a symbolic manner, in contrast to concrete execution. Some values are represented as symbolic values, which can take on different concrete values.

To select the appropriate verification approach, one can take several factors into account. The impact of a bug, the likelihood that a bug occurs, and the available budget are common considerations. Comparing two extremes, it is non-problematic to have an image on a website misaligned by a pixel opposed to the crash of the Ariana 5 Flight 501, costing hundreds of millions of dollars [19].

The main problem with automated approaches is that some constructs regularly found in programming languages increase the number of possible inputs and the search space by an extreme amount, known as the state explosion problem and the path explosion problem.[1] Major contributors to the path explosion problem are: iteration structures and recursion, concurrency, exceptions and reference aliasing. Ongoing research on automated approaches, a part of the

[1]The variant that is used depends on the context of the automated approach. The state explosion problem is regularly used in the context of model checking and the path explosion problem is regularly used in the context of symbolic execution. In this thesis, the path explosion problem definition will be used.

field of formal methods, aims to reduce the effects caused by the path explosion problem while maintaining the capability to find as many bugs as possible.

We designed and implemented an automated verification approach based on symbolic execution, including several optimizations. It aims to reduce the effects caused by the path explosion as a result of: branching statements, concurrency and reference aliasing. The target language of our approach is the OOX language, a language we designed for the purpose of this thesis. OOX is an intermediate verification language that is used as a core language to model object-oriented languages with support for concurrency, such as Java and C#.

The main contributions of this thesis are:

1. The OOX language: a new intermediate verification language, including its formal static- and dynamic semantics;
2. A generalization of symbolic execution to concurrent programs;
3. A formalization of applying partial order reduction to this generalized approach of symbolic execution; and
4. The implementation of our symbolic execution engine, including several promising optimization techniques.

Thesis Outline. We begin this thesis with the preliminaries in Chapter 2. We present the OOX language in Chapter 3 and continue with the formalization of symbolic execution and the optimizations in Chapter 4. Its effectiveness will be evaluated in Chapter 5. We will then present the related work in Chapter 6 and conclude and present future work in Chapter 7.

Chapter 2

Preliminary

Sets, Sequences and Mappings. We assume that the reader is familiar with the notion of sets and sequences. We use $\mathcal{P}(A)$ to denote the power set of the set A. We use the binary set operator $A \uplus B$ to denote an update of B inside A. That is, the union of A and B where B is first subtracted

$$A \uplus B = (A - B) \cup B$$

We use the notion of a mapping for key-value pairs.

Definition 1 (Mapping)**.** Let a mapping M be defined as

$$M[k \mapsto v](x) = \begin{cases} M(x) & \textbf{if } x \neq k \\ v & \textbf{if } x = k \end{cases}$$

Let [] denote an empty mapping.

Formal Verification. A program path is an unique sequence of statements of a program. A program can be represented as a set of program paths, which is of size at least one and is potentially infinite. Any non-trivial program has statements which cause branching behaviour. One of such statements is the if-then-else statement. For example, the statement `if E then ... else ...` has two program paths. One in which `E` holds true and one in which not `E` holds true. Such conditions are expressed using `assume E;` statements, which instruct the verification back-end to assume that the rest of the program paths can only be executed, or are feasible, when the assumption is true. `assert E;` statements are used to verify whether the assertion `E` is valid.

Pseudocode Notation. We use the pseudocode notation that is common in the literature. We introduce one new operation: "inner access", which is denoted by $a \bullet b$. We use this operation in the context of state update. For example, when we update the call stack σ of the thread $\mathcal{T}_{tid}$ to some σ', we write $\mathcal{T}_{tid} \bullet \sigma \leftarrow \sigma'$. We let an update of a thread $\mathcal{T}_{tid} \in \mathcal{T}$ include the update inside $\mathcal{T}$, meaning that when executing the update from above, the statement $\mathcal{T} \leftarrow (\mathcal{T} - \{\mathcal{T}_{tid}\}) \cup \{\mathcal{T}_{tid}\}$ is implicit. We use – to denote a comment.

Auxiliary Notation. We use $\perp$ to denote that some operation is undefined. For example, in the dynamic semantics of expression evaluation, when evaluating division by zero

$$\mathcal{E}[\![binop(E_1, /, E_2)]\!](\sigma) = \begin{cases} \perp & \textbf{if } \mathcal{E}[\![E_2]\!](\sigma, h) = 0 \\ \mathcal{E}[\![E_1]\!](\sigma, h) / \mathcal{E}[\![E_2]\!](\sigma, h) & \text{otherwise} \end{cases}$$

We sometimes use an annotated variant $\perp_h$. We let $f^{\circ n}(x)$ denote n function compositions of f. For example

$$f^{\circ 2}(x) = (f \circ f)(x)$$

When defining an explicit example, we use $\square$ to denote the end of an example.

Chapter 3

The OOX Language

OOX is a language designed to be an intermediate verification language that can represent object-oriented programming languages like Java and C#. The language has support for classes and objects as first-class citizens, and has a strong type system. Another main feature is the support for concurrency via a fork-join model, including synchronization using a lock mechanism.

In its current state, the object-oriented language features are limited to declaring classes and using objects. The three main features associated with object-oriented languages: inheritance, encapsulation and polymorphism are outside the scope of this thesis. As such, the language can be viewed, not as object-oriented, but as a language with structs. However, the current design lays the foundation to extend the language with such features.

Example 1 (Concurrent Counting)**.** Suppose a `Counter` class is defined, which supports incrementing its internal counter `count` safely in a concurrent setting and suppose a `count` method is defined. This `count` method takes an `initial` value and increments the counter `N` times, each in a separate thread. Finally, the aim is to verify that `count` is equal to `initial + N`.

```
1  class Counter
2  {
3      int current;
4      Counter(int initial) { this.current := initial; }
5      static void count(int initial)
6          requires(initial >= 0)
7          exceptional(false)
8      {
9          Counter counter := new Counter(initial);
10         int i := 0;
11         while (i < N) {
12             fork counter.increment();
13             i := i + 1;
14         }
15         join;
16         int value := counter.current;
17         assert value == initial + N;
18     }
19     void increment()
20     {
21         lock (this)
22         {
23             int value := this.count;
24             this.current := value + 1;
25         }
26     }
27 }
```

3.1 The Memory

The memory of an OOX program consists of two distinct parts: the stack and the heap. The stack contains the arguments and the local variable declarations, each living in a stack frame corresponding to the scope they live in. The stack frames are allocated and deallocated automatically when a new scope is entered or exited. The heap contains the variables that are created using the `new` operator. Object structures on the are accesses via references. A variable on the stack can be reference, allowing it to access object structures in the heap. The state of a program is defined as a pair containing both a stack and a heap.

3.1.1 The Stack

Definition 2 (The Stack)**.** The stack, denoted by σ, is a sequence of stack frames where the top-most stack frame is the stack frame of the current scope.

Four operations are defined on stacks: reading from a stack, writing to a stack, pushing a stack frame on a stack and popping a stack frame from a stack. Let $\sigma(x)$ denote reading from the stack, defined as

$$\sigma(x) = \sigma_1(x) \textbf{ where } \sigma = [\sigma_1, \ldots, \sigma_n]$$

and let $\sigma[x \mapsto v]$ denote writing to the stack, defined as

$$\sigma[x \mapsto v] = \sigma_1[x \mapsto v] \textbf{ where } \sigma = [\sigma_1, \ldots, \sigma_n]$$

Let $push(\sigma)$ and $pop(\sigma)$ be the two operations that allow for stack frame management. That is,

$$push(\sigma) = [[], \sigma_1, \ldots, \sigma_n] \textbf{ where } \sigma = [\sigma_1, \ldots, \sigma_n]$$
$$pop(\sigma) = (\sigma_1, \ldots, \sigma_n] \textbf{ where } \sigma = [\sigma_1, \ldots, \sigma_n]$$

Definition 3 (A Stack Frame)**.** Let a stack frame be defined as a mapping from identifiers to values.

3.1.2 The Heap

Definition 4 (The Heap)**.** The heap, denoted by h, is a mapping from references to object structures. An object structure, denoted by $\mathcal{O}$, is a mapping from fields to values.

The heap has support for three operations, allocation, reading and writing, where the latter two are inherited from the mapping structure. Allocation is defined by $alloc(h)$, which returns a fresh unique reference. Reading from a heap, denoted by $h(ref_x)$, retrieves the object structure of the reference ref_x in h iff x is defined h and not null. When ref_x is either not defined in h or is `null`, $\bot_h$ is returned. Writing to a heap, denoted by $h[ref_x \mapsto \mathcal{O}]$, returns an updated heap in which the reference ref_x maps to the object structure $\mathcal{O}$.

3.2 The Semantics

The semantics of OOX is defined by both the static- and dynamic semantics.

3.2.1 The Static Semantics

The static semantics are defined using inference rules. They describe the well-formedness, using the **ok** and **ok in** operators, and type judgements, as described in Section 3.4.

3.2.2 The Dynamic Semantics

The dynamic semantics are defined using a big-step operational semantics for the execution of the program, and a small-step operational semantics for the execution of statements and methods calls. The execution of expressions is described using a denotational semantics, see Section 3.8.9 for more details.

Program Transitions. Program transitions denote a reduction step in the execution of a program.

$$(\underset{prog}{\longrightarrow}) : CompilationUnit \times Locks \times Threads \times Heap \rightarrow Locks \times Threads \times Heap$$

Statement Transitions. Statement transitions denote a reduction step in the execution of a statement.

$$(\underset{stat}{\longrightarrow}) : Statement \times (Stack \times Heap) \rightarrow Statement \times (Stack \times Heap)$$

Left-Hand Side Transitions. Left-hand side transitions denote a reduction step in the execution of the left-hand side of an assignment.

$$(\underset{lhs}{\longrightarrow}) : Lhs \times Value \times (Stack \times Heap) \rightarrow (Stack \times Heap)$$

Right-Hand Side Transitions. Right-hand side transitions denote a reduction step in the execution of the right-hand side of an assignment.

$$(\underset{rhs}{\longrightarrow}) : Rhs \times (Stack \times Heap) \rightarrow Value \times Heap$$

Invocation Transitions. Invocation transitions denote a reduction step in the execution of a method or constructor invocation.

$$(\underset{inv}{\longrightarrow}) : Invocation \times (Stack \times Heap) \rightarrow Statement \times (Stack \times Heap)$$

3.3 The Abstract Syntax

Below is the complete list of the various syntactical categories that make the abstract syntax of OOX.

$$
\begin{array}{rl}
U \in \mathit{CompilationUnit} & n \in \mathit{Nat} \\
M \in \mathit{Method} & z \in \mathit{Int} \\
K \in \mathit{Constructor} & r \in \mathit{Real} \\
\varphi \in \mathit{Specification} & b \in \mathit{Bool} \\
P \in \mathit{FormalParameter} & s \in \mathit{String} \\
F \in \mathit{Field} & c \in \mathit{Char} \\
S \in \mathit{Statement} & \oplus \in \mathit{UnaryOperator} \\
t \in \mathit{Lhs} & \otimes \in \mathit{BinaryOperator} \\
v \in \mathit{Rhs} & \omega \in \mathit{Type} \\
I \in \mathit{Invocation} & \tau \in \mathit{NonVoidType} \\
E \in \mathit{Expression} & i \in \mathit{Identifier}
\end{array}
$$

3.4 Types

OOX has a strong static type system where every expression, assignment target, assignment value and parameter has a type. The type rules of OOX are enforced in the static semantics of the language.

Definition 5 (The Type Environment)**.** The type environment, denoted by Γ, contains the types of the variables in the current context.

The following three operations are defined on type environments

Type Judgement. Let $\Gamma \vdash E : \omega$ denote type judgment, which expresses that E has type ω when evaluated under the type environment Γ.

Type Existence. Let $x : \omega \in \Gamma$ denote type existence, which expresses that there exists a variable x in Γ and is of type ω. Let $x : \omega \notin \Gamma$ denote that there does not exist a variable x of type ω in Γ.

Type Insertion. Let $\Gamma \vdash x : \omega$ **ok in** X denote type insertion, which expresses that the variable x has type ω in the type environment Γ when evaluated within X.

There are two kind of types defined: types and non-void types.

$$
\begin{array}{lll}
\omega \in \mathit{Type} & ::= & \mathit{type}(\tau) \mid \texttt{void} \\
\tau \in \mathit{NonVoidType} & ::= & \texttt{uint} \mid \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{char} \mid \mathit{ref}(C) \mid \mathit{array}(\tau)
\end{array}
$$

Each non-void type has an associated default value, defined by

$$default : NonVoidType \to Expression$$

where

$$\begin{aligned} default(\texttt{uint}) &= 0 & default(\texttt{string}) &= \texttt{null} \\ default(\texttt{int}) &= 0 & default(ref(C)) &= \texttt{null} \\ default(\texttt{float}) &= 0.0 & default(array(\tau)) &= \texttt{null} \\ default(\texttt{bool}) &= \texttt{false} \end{aligned}$$

3.4.1 Primitive and Reference Types

There are two categories of types: primitive types and reference types. Primitive types are values which do not point to some something allocated on the heap. Let `uint`, `int`, `float`, `bool` and `char` be the primitive types. Reference types are references to a value allocated on the heap. Let `string`, $ref(C)$ and $array(\tau)$ be the reference types.

3.4.2 Subtyping

OOX limited support for subtyping. Subtyping in the form of inheritance cannot be defined by the user. Some (sub)expressions have support for subtyping, more specifically operations with numbers and references. Let $\tau \prec \tau'$ denote the subtype relation where τ' is a subtype of τ.

There are two subtypes defined: *REF*, of which every reference type is a subtype, and *NUM* of which every numerical type is a subtype. These subtypes cannot be used directly by the user. The subtype *REF* allows `null` to be polymorphic. For example, to allow that the expression `null == x` is type correct. The subtype *NUM* serves a similar purpose for expressions of numerical types, e.g. `0.0 + 1` where `0.0` is of type `float` and `1` is type `int`.

3.5 Compilation Units

A compilation unit is the root node of the Abstract Syntax Tree (AST). It defines a complete OOX program.

$$U \in CompilationUnit ::= program(\underline{C})$$

3.5.1 Threads and Scheduling

A program consists of one or more threads. Each thread executes a part of the program and has its own stack. New threads can be created using fork statements, and can be synchronized using both join statements and lock statements. All threads share the same heap, the heap of the program. The threads are represented as a set, denoted by $\mathcal{T}$. Let $\mathcal{T}_{tid}$ denote a thread with thread id $tid \in \mathbb{N}$, defined as a pair (S, σ), consisting of the next statement to be executed and the stack of that thread.

Let $\Sigma : \mathcal{P}(Thread) \to \mathbb{N}$ denote a scheduling function that chooses the next thread id to be executed from $\mathcal{T}$. No fairness assumptions are made about Σ.

3.5.2 The Semantics

A program is valid in the static semantics when a static `main` function is defined. This function is required to have return type `void` and have no formal parameters.

$$\frac{\exists class(\underline{M}, \_, \_) \in \underline{C} : \exists method(b, \omega, i, \_, \_, \_) \in \underline{M} : b = \texttt{true} \wedge \omega = \texttt{void} \wedge i = \texttt{main}}{\Gamma \vdash program(\underline{C})\ \textbf{ok}}$$

(SSEM - program)

Program Initialization and Termination. The execution of the program begins at the `main` method. Let `running'` be a special reference in the lock set to denote that the program has started the execution.

$$\frac{\begin{array}{c}\exists C = class(\underline{M}, \_, \_) \in \underline{C} : \exists M = method(\_, \_, i, \_, \_, \_) \in \underline{M} : i = \texttt{main} \\ \mathcal{T}_{tid} = (call(invoke(C, M, [])), [])\end{array}}{\langle program(\underline{C}), \emptyset, \emptyset, [] \rangle \xrightarrow[prog]{} \langle \{\texttt{running'}\}, \{\mathcal{T}_{tid}\}, h \rangle}$$

(DSEM - program: initialization)

The execution of the program has finished when it has started and there are no more threads to be executed.

$$\frac{\texttt{running'} \in \mathcal{L}}{\langle U, \mathcal{L}, \emptyset, h \rangle \xrightarrow[prog]{} \langle \mathcal{L}, \emptyset, h \rangle}$$

(DSEM - program: termination)

Thread Spawning and Termination. When the next statement to be executed is a `fork` statement, a new thread is spawned and the forking thread continues its execution.

$$\frac{\begin{array}{c}tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (seq(fork(I), S_2), \sigma) \in \mathcal{T} \\ \mathcal{T}_{fresh} = (call(I), \sigma) \\ \mathcal{T}'_{tid} = (S_2, \sigma)\end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h \rangle \xrightarrow[prog]{} \langle \mathcal{L}, \mathcal{T} \uplus \{\mathcal{T}'_{tid}, \mathcal{T}_{fresh}\}, h \rangle}$$

(DSEM - program: fork statement)

A thread will terminate when the single remaining statement of that thread is a `skip` statement. This will results in the thread being removed from the set of threads.

$$\frac{\begin{array}{c}tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (\texttt{skip}, \sigma) \in \mathcal{T}\end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h \rangle \xrightarrow[prog]{} \langle \mathcal{L}, \mathcal{T} - \{\mathcal{T}_{tid}\}, h \rangle}$$

(DSEM - program: thread termination)

Lock Acquisition and Releasing. When the next statement to be executed is a `lock` statement, there are two options: the value of variable i is in the lock set $\mathcal{L}$ or it is not in the lock set. When the value of variable i is not in the lock set, the reference is added to the lock set and the

execution continues. When the value of i is in the lock set, the thread performs no operation.

$$\frac{\begin{array}{c} tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (seq(lock(i), S_2), \sigma) \in \mathcal{T} \\ \sigma(i) = ref_n \notin \mathcal{L} \\ \mathcal{T}'_{tid} = (S_2, \sigma) \end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h\rangle \xrightarrow[prog]{} \langle \mathcal{L} \cup \{ref_n\}, \mathcal{T} \uplus \{\mathcal{T}'_{tid}\}, h\rangle} \quad \text{(DSEM - program: lock statement (unlocked))}$$

$$\frac{\begin{array}{c} tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (seq(lock(i), S_2), \sigma) \in \mathcal{T} \\ \sigma(i) = ref_n \in \mathcal{L} \end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h\rangle \xrightarrow[prog]{} \langle \mathcal{L}, \mathcal{T}, h\rangle} \quad \text{(DSEM - program: lock statement (locked))}$$

$$\frac{\begin{array}{c} tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (seq(lock(i), S_2), \sigma) \in \mathcal{T} \\ \texttt{null} = \sigma(i) \end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h\rangle \xrightarrow[prog]{} \langle \bot_h\rangle} \quad \text{(DSEM - program: lock statement (null dereference))}$$

When the next statement to be executed is an `unlock` statement, the value of the variable i is removed from the lock set $\mathcal{L}$ and the thread continues its execution.

$$\frac{\begin{array}{c} tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (seq(unlock(i), S_2), \sigma) \in \mathcal{T} \\ ref_n = \sigma(i) \\ \mathcal{T}'_{tid} = (S_2, \sigma) \end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h\rangle \xrightarrow[prog]{} \langle \mathcal{L} - \{ref_n\}, \mathcal{T} \uplus \{\mathcal{T}'_{tid}\}, h\rangle} \quad \text{(DSEM - program: unlock statement)}$$

Thread Joining. When the next statement to be executed is a `join` statement, there are two options: the first option is that there exists a child thread of the current thread, meaning that no operation will be performed. The second option is that there does not exist a child thread of the current thread, meaning that the execution continues.

$$\frac{\begin{array}{c} tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (seq(\texttt{join}, S_2), \sigma) \in \mathcal{T} \\ \exists \mathcal{T}_{tid'} \in \mathcal{T} : parent(\mathcal{T}_{tid'}) = tid \end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h\rangle \xrightarrow[prog]{} \langle \mathcal{L}, \mathcal{T}, h\rangle} \quad \text{(DSEM - program: join statement (wait))}$$

$$\frac{\begin{array}{c} tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (seq(\texttt{join}, S_2), \sigma) \in \mathcal{T} \\ \forall \mathcal{T}_{tid'} \in \mathcal{T} : parent(\mathcal{T}_{tid'}) \neq tid \\ \mathcal{T}'_{tid} = (S_2, \sigma) \end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h\rangle \xrightarrow[prog]{} \langle \mathcal{L}, \mathcal{T} \uplus \{\mathcal{T}'_{tid}\}, h\rangle} \quad \text{(DSEM - program: join statement (continue))}$$

Exceptional Termination. When a thread results in an exceptional state, the program halts and terminates in the same exceptional state.

$$\frac{\begin{array}{c} tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (S, \sigma) \in \mathcal{T} \\ \langle S, (\sigma, h)\rangle \xrightarrow[stat]{} \langle \bot_{h'} \rangle \end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h\rangle \xrightarrow[prog]{} \langle \bot_{h'} \rangle} \qquad \text{(DSEM - program: exceptional execution)}$$

Statement Execution. When none of the above cases match, the thread executes the next statement.

$$\frac{\begin{array}{c} tid = \Sigma(\mathcal{T}) \\ \mathcal{T}_{tid} = (S, \sigma) \in \mathcal{T} \\ \langle S, (\sigma, h)\rangle \xrightarrow[stat]{}^{1} \langle S', (\sigma', h')\rangle \\ \mathcal{T}'_{tid} = (S', \sigma') \end{array}}{\langle U, \mathcal{L}, \mathcal{T}, h\rangle \xrightarrow[prog]{} \langle \mathcal{L}, \mathcal{T} \uplus \{\mathcal{T}'_{tid}\}, h'\rangle} \qquad \text{(DSEM - program: normal execution)}$$

3.6 Classes

A class is a structure containing members. There are three kind of members: methods, constructors and fields.

$$C \in \mathit{Class} ::= \mathit{class}(\underline{M}, \underline{K}, \underline{F})$$

3.6.1 Methods and Constructors

A method is an executable part of a program, defined within a class. It consists of a sequence of statements which can be invoked via a method call.

The type definition of a method consists of both the return type and zero or more formal parameters. When the return type of a method is not `void`, a call to the method results in a value. The type of the return value must match the return type of the method.

A method can be either static or non-static. The difference between the two is that a non-static method is invoked on an object, which is passed on to the method body as an implicit formal parameter. This implicit formal parameter is named `this` and refers to the object on which the method is invoked.

Constructors are a special kind of static methods. They are used to create a new instance of a class. Constructors, unlike regular static methods, have an implicit `this` formal parameter. This formal parameter refers to a newly allocated object.

$$\begin{array}{ll} M \in \mathit{Method} & ::= \mathit{method}(b, \omega, i, \varphi, \underline{P}, S) \\ K \in \mathit{Constructor} & ::= \mathit{constructor}(C, \varphi, \underline{P}, S) \end{array}$$

Specifications. Methods and constructors can be annotated with a specification. Specifications consist of a pre-condition, a post-condition and an exceptional post-condition. Such specifications can be used by the verification back-end to verify whether the method or constructor satisfies its specification.

$$\varphi \in \mathit{Specification} ::= \mathit{specification}(E_1, E_2, E_3)$$

$$\frac{\Gamma \vdash E_1 : \texttt{bool} \quad \Gamma \vdash E_2 : \texttt{bool} \quad \Gamma \vdash E_3 : \texttt{bool}}{\Gamma \vdash \mathit{specification}(E_1, E_2, E_3)\ \textbf{ok}} \qquad \text{(SSEM - specification)}$$

Formal Parameters. Formal parameters are a part of the type definition of methods and constructors. They define which arguments must be passed to the method or constructor when invoked. The formal parameters can be used as local variables inside the body of the method or constructor.

$$P \in \mathit{FormalParameter} ::= \mathit{param}(\tau, i)$$

3.6.2 Fields

Fields are members of a class which make up the data of classes and objects. Fields can be either static or non-static. Static fields are global variables and non-static fields are variables of a specific object.

$$F \in \mathit{Field} ::= \mathit{field}(b, \tau, i)$$

3.7 Statements

Below is the complete list of statements in the abstract syntax of OOX.

$$
\begin{aligned}
S \in \mathit{Statement} ::= {} & \mathit{declare}(\tau, i) \\
\mid {} & \mathit{assign}(t, v) \\
\mid {} & \mathit{call}(I) \\
\mid {} & \texttt{skip} \\
\mid {} & \mathit{assert}(E) \\
\mid {} & \mathit{assume}(E) \\
\mid {} & \mathit{while}(E, S) \\
\mid {} & \mathit{ite}(E, S_1, S_2) \\
\mid {} & \texttt{continue} \\
\mid {} & \texttt{break} \\
\mid {} & \texttt{return} \\
\mid {} & \mathit{return}(E) \\
\mid {} & \texttt{throw} \\
\mid {} & \mathit{try}(S_1, S_2) \\
\mid {} & \mathit{block}(S) \\
\mid {} & \mathit{lock}(i) \\
\mid {} & \mathit{unlock}(i) \\
\mid {} & \texttt{join} \\
\mid {} & \mathit{fork}(I) \\
\mid {} & \mathit{seq}(S_1, S_2) \\
\mid {} & \texttt{pop}
\end{aligned}
$$

3.7.1 Variable Declarations

Variable declarations introduce a new variable, which are declared in the stack. The default value of the corresponding type will be assigned to the variable.

$$\mathit{declare} : \mathit{NonVoidType} \times \mathit{Identifier} \to \mathit{Statement}$$

Variable declaration are valid in the static semantics if there exist no variable with the same name, i.e. variable shadowing is not allowed.

$$\frac{i : \tau \notin \Gamma}{\Gamma \vdash \mathit{declare}(\tau, i)\ \mathbf{ok}} \qquad \text{(SSEM - variable declaration)}$$

$$\frac{}{\langle \mathit{declare}(\tau, i), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \texttt{skip}, (\sigma[i \mapsto \mathit{default}(\tau)], h)\rangle} \qquad \text{(DSEM - variable declaration)}$$

3.7.2 Assignment Statements

Assignment statements update the value of a variable, field or array element. An assignment consists of two parts: the left-hand side and the right-hand side.

$$assign : Lhs \times Rhs \to Statement$$

Assignment statements are valid in the static semantics if the types of the left-hand side and the right-hand side match.

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash v : \tau}{\Gamma \vdash assign(t, v)\ \mathbf{ok}} \qquad \text{(SSEM - assignment)}$$

$$\frac{\begin{array}{c} v = rhs_{call}(I) \\ v' = rhs_{expr}(var(retval')) \\ S' = seq(call(I), assign(t, v')) \end{array}}{\langle assign(t, v), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle S', (\sigma, h)\rangle} \qquad \text{(DSEM - assignment: invocation rhs)}$$

$$\frac{\begin{array}{c} \langle v, (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle v', h'\rangle \\ \langle t, v', (\sigma, h')\rangle \underset{lhs}{\longrightarrow} \langle (\sigma'', h'')\rangle \end{array}}{\langle assign(t, v), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \texttt{skip}, (\sigma'', h'')\rangle} \qquad \text{(DSEM - assignment: non-invocation rhs)}$$

$$\frac{\langle v, (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle \bot_{h'}\rangle}{\langle assign(t, v), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \bot_{h'}\rangle} \qquad \text{(DSEM - assignment: exception rhs)}$$

$$\frac{\begin{array}{c} \langle v, (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle v', h'\rangle \\ \langle t, v', (\sigma, h')\rangle \underset{lhs}{\longrightarrow} \langle \bot_{h''}\rangle \end{array}}{\langle assign(t, v), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \bot_{h''}\rangle} \qquad \text{(DSEM - assignment: exception lhs)}$$

Left-Hand Side

The left-hand side of an assignment determines the memory location in which the value will be written. The left-hand side can be either: a variable, a field of an object or an element of an array.

$$\begin{aligned} t \in Lhs ::= {} & lhs_{var}(i) \\ \mid {} & lhs_{field}(i, F) \\ \mid {} & lhs_{elem}(i, E) \end{aligned}$$

Variable Target. Variable targets allow for writing a value to a variable declared on the stack. For example, in the assignment `x := e;`, the expression `e` is written to the variable target `x`.

$$lhs_{var} : Identifier \to Lhs$$

A variable target is valid in the static semantics when the variable is declared.

$$\frac{i : \tau \in \Gamma}{\Gamma \vdash lhs_{var}(i) : \tau} \quad \text{(SSEM - variable target)}$$

$$\frac{}{\langle lhs_{var}(i), v, (\sigma, h)\rangle \xrightarrow[lhs]{} \langle (\sigma[i \mapsto v], h)\rangle} \quad \text{(DSEM - variable target)}$$

Object Field Target. Object field targets allow for targeting a field of an object. For example, in the assignment `x.f := e;`, the expression `e` is written to the object field target `x.f` of which `f` is the field and `x` is the variable.

$$lhs_{field} : Identifier \times Field \to Lhs$$

An object field target is valid in the static semantics when the variable is declared and its type is a class. This class must contain the field that is targeted.

$$\frac{\begin{array}{c} i : \tau \in \Gamma \\ \tau = ref(class(\_, \_, \underline{F})) \\ F = field(\tau', \_) \in \underline{F} \end{array}}{\Gamma \vdash lhs_{field}(i, F) : \tau'} \quad \text{(SSEM - field target)}$$

$$\frac{\sigma(i) = ref_x \quad h(ref_x) = \mathcal{O} \quad F = field(\_, i')}{\langle lhs_{field}(i, F), v, (\sigma, h)\rangle \xrightarrow[lhs]{} \langle (\sigma, h[ref_x \mapsto \mathcal{O}[i' \mapsto v]])\rangle} \quad \text{(DSEM - field target)}$$

$$\frac{\sigma(i) = \texttt{null}}{\langle lhs_{field}(i, F), v, (\sigma, h)\rangle \xrightarrow[lhs]{} \langle \bot_h\rangle} \quad \text{(DSEM - field target: null dereference)}$$

Array Element Target. Array element targets allow for targeting an element of an array. For example, in the assignment `a[i] := e;`, the expression `e` is written to the array element target `a[i]` of which `a` is the variable and `[i]` is the specific element that is targeted.

$$lhs_{elem} : Identifier \times Expression \to Lhs$$

An array element target is valid in the static semantics when the index is of type `int` and the variable is a declared variable of type *array*.

$$\frac{\begin{array}{c} \Gamma \vdash E : \texttt{int} \\ i : array(\tau) \in \Gamma \end{array}}{\Gamma \vdash lhs_{elem}(i, E) : \tau} \quad \text{(SSEM - element target)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![E]\!](\sigma,h) = e \neq \bot \\ \sigma(i) = ref_x \\ h(ref_x) = \mathcal{O} = [elem_0 \mapsto v_0, \ldots, elem_n \mapsto v_n) \\ 0 \leq e < n\end{array}}{\langle lhs_{elem}(i,E), v, (\sigma,h)\rangle \underset{lhs}{\longrightarrow} \langle(\sigma, h[ref_x \mapsto \mathcal{O}[elem_e \mapsto v]])\rangle} \quad \text{(DSEM - element target)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![E]\!](\sigma,h) = e \neq \bot \\ \sigma(i) = ref_x \\ h(ref_x) = \mathcal{O} = [elem_0 \mapsto v_0, \ldots, elem_n \mapsto v_n) \\ e < 0 \vee e \geq n\end{array}}{\langle lhs_{elem}(i,E), v, (\sigma,h)\rangle \underset{lhs}{\longrightarrow} \langle \bot_h\rangle} \quad \text{(DSEM - element target: outside bounds)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma,h) = \bot}{\langle lhs_{elem}(i,E), v, (\sigma,h)\rangle \underset{lhs}{\longrightarrow} \langle \bot_h\rangle} \quad \text{(DSEM - element target: evaluation exception)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![E]\!](\sigma,h) = e \neq \bot \\ \sigma(i) = \texttt{null}\end{array}}{\langle lhs_{elem}(i,E), v, (\sigma,h)\rangle \underset{lhs}{\longrightarrow} \langle \bot_h\rangle} \quad \text{(DSEM - element target: null dereference)}$$

Right-Hand Side

The right-hand side of an assignment determines the value that will be assigned to the left-hand side. The right-hand side can be either: an expression, a field of an object, a method- or constructor call, an array element or the instantiation of a new array.

$$\begin{array}{rcl} v \in Rhs & ::= & rhs_{expr}(E) \\ & \mid & rhs_{field}(i,F) \\ & \mid & rhs_{call}(I) \\ & \mid & rhs_{elem}(i,E) \\ & \mid & rhs_{array}(\tau, \underline{E}) \end{array}$$

Expression Values. Expression values are used to assign the evaluated expression to the left-hand side. For example, in the assignment `x := y + 1;`, the evaluated expression `y + 1` will be assigned to the left-hand side.

$$rhs_{expr} : Expression \to Rhs$$

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash rhs_{expr}(E) : \tau} \quad \text{(SSEM - expression value)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = v \neq \bot}{\langle rhs_{expr}(E), (\sigma, h)\rangle \xrightarrow[lhs]{} \langle v, h\rangle} \quad \text{(DSEM - expression value)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = \bot}{\langle rhs_{expr}(E), (\sigma, h)\rangle \xrightarrow[lhs]{} \langle \bot_h\rangle} \quad \text{(DSEM - expression value: evaluation exception)}$$

Field Values. Field values are used to assign the value of a field of an object to the left-hand side. For example, in the assignment `x := y.f;`, the evaluated field value `y.f` of field `f` of variable `y` will be assigned to the left-hand side.

$$rhs_{field} : Identifier \times Field \to Rhs$$

A field value is valid in the static semantics when the variable is declared and its type is a class. This class must contain the field that is targeted.

$$\frac{\begin{array}{c} i : ref(class(\_, \_, \underline{F})) \in \Gamma \\ F = field(\tau, \_) \in \underline{F} \end{array}}{\Gamma \vdash rhs_{field}(i, F) : \tau} \quad \text{(SSEM - field value)}$$

$$\frac{\sigma(i) = ref_x \quad h(ref_x) = \mathcal{O} \quad \mathcal{O}(i') = v}{\langle rhs_{field}(i, field(\_, i')), (\sigma, h)\rangle \xrightarrow[rhs]{} \langle v, h\rangle} \quad \text{(DSEM - field value)}$$

$$\frac{\sigma(i) = \texttt{null}}{\langle rhs_{field}(i, F), (\sigma, h)\rangle \xrightarrow[rhs]{} \langle \bot_h\rangle} \quad \text{(DSEM - field value: null dereference)}$$

Call Result Values. Call result values are used to assign the return value of the method or constructor call to the left-hand side. For example, in the assignment `x := Math.min(a, b);`, the return value of the method call `Math.min(a, b)` will be assigned to the left-hand side.

$$rhs_{call} : Invocation \to Rhs$$

A call result value is valid in the static semantics when the invocation I is valid in the static semantics. See Subsection 3.7.3 for more information.

$$\frac{\Gamma \vdash I : type(\tau)}{\Gamma \vdash rhs_{call}(I) : \tau} \quad \text{(Static semantics of call result value)}$$

Note that the dynamic semantics of call result values are handled in the dynamic semantics of assignment itself. See Subsection 3.7.2 for more information.

Array Element Values. Array element values are used to assign an element of an array to the left-hand side. For example, in the assignment `x := a[i];`, the evaluated array element `a[i]` of element `i` of array `a` will be assigned to the left-hand side.

$$rhs_{elem} : Identifier \times Expression \to Rhs$$

An array element value is valid in the static semantics when the index expression is of type `int` and the variable is an existing variable of `array` type.

$$\frac{\Gamma \vdash E : \texttt{int} \quad i : array(\tau) \in \Gamma}{\Gamma \vdash rhs_{elem}(i, E) : \tau} \qquad \text{(SSEM - array element value)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = e \neq \bot \quad \sigma(i) = ref_x \quad h(ref_x) = \mathcal{O} = [elem_0 \mapsto v_0, \ldots, elem_n \mapsto v_n) \quad 0 \leq e < n}{\langle rhs_{elem}(i, E), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle v_e, h\rangle} \qquad \text{(DSEM - array element value)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = e \neq \bot \quad \sigma(i) = ref_x \quad h(ref_x) = [elem_0 \mapsto v_0, \ldots, elem_n \mapsto v_n) \quad e < 0 \vee e \geq n}{\langle rhs_{elem}(i, E), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle \bot_h\rangle} \qquad \text{(DSEM - array element value: outside bounds)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = \bot}{\langle rhs_{elem}(i, E), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle \bot_h\rangle} \qquad \text{(DSEM - array element value: evaluation exception)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = e \neq \bot \quad \sigma(i) = \texttt{null}}{\langle rhs_{elem}(i, E), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle \bot_h\rangle} \qquad \text{(DSEM - array element value: null dereference)}$$

Array Instantiation. Array instantiation can be used to assign a newly allocated array to the left-hand side. For example, in the assignment `x := new int[3];`, a new array of size `3` containing elements of type `int` will be assigned to the left-hand side. There is support for both single- and multidimensional arrays.

$$rhs_{array} : NonVoidType \times [Expression] \to Rhs$$

An array instantiation is valid in the static semantics when each size is of type `int` and is at least one-dimensional.

$$\frac{|\underline{E}| \geq 1 \quad \Gamma \vdash \underline{E}_1 : \texttt{int} \quad \ldots \quad \Gamma \vdash \underline{E}_n : \texttt{int}}{\Gamma \vdash rhs_{array}(\tau, \underline{E}) : array^{\circ n}(\tau)} \qquad \text{(SSEM - array instantiation)}$$

$$\frac{\begin{array}{c}|\underline{E}| > 1 \\ alloc(h) = ref_{fresh} \\ 0 < s = \mathcal{E}[\![\underline{E}_1]\!](\sigma, h) \neq \bot \\ \underline{E}' = [\underline{E}_2, \ldots, \underline{E}_n] \\ \tau = array(\tau') \\ \langle rhs_{array}(\tau', \underline{E}'), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle ref_0, h_0\rangle \quad \ldots \quad \langle rhs_{array}(\tau', \underline{E}'), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle ref_{s-1}, h_{s-1}\rangle \\ h' = \cup_{i=0}^{s-1} h_i \\ \mathcal{O} = [elem_0 \mapsto ref_0, \ldots, elem_s \mapsto ref_s)\end{array}}{\langle rhs_{array}(\tau, \underline{E}), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle ref_{fresh}, h'[ref_{fresh} \mapsto \mathcal{O}]\rangle}$$

(DSEM - array instantiation: multi-dimensional)

$$\frac{\begin{array}{c}|\underline{E}| = 1 \\ alloc(h) = ref_{fresh} \\ 0 < s = \mathcal{E}[\![\underline{E}_1]\!](\sigma, h) \neq \bot \\ \mathcal{O} = [elem_0 \mapsto default(\tau), \ldots, elem_{s-1} \mapsto default(\tau)]\end{array}}{\langle rhs_{array}(\tau, \underline{E}), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle ref_{fresh}, h[ref_{fresh} \mapsto \mathcal{O}]\rangle}$$

(DSEM - array instantiation: one dimension)

$$\frac{\mathcal{E}[\![\underline{E}_1]\!](\sigma, h) = \bot}{\langle rhs_{array}(\tau, \underline{E}), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle \bot_h\rangle}$$

(DSEM - array instantiation: evaluation exception)

$$\frac{\exists E \in \underline{E} : \mathcal{E}[\![E]\!](\sigma, h) \leq 0}{\langle rhs_{array}(\tau, \underline{E}), (\sigma, h)\rangle \underset{rhs}{\longrightarrow} \langle \bot_h\rangle}$$

(DSEM - array instantiation: non-positive size)

3.7.3 Method Call Statements

Method call statements provide a way to execute the statements of a static or non-static method with the formal parameters set the arguments.

$$call : Invocation \rightarrow Statement$$

$$\frac{\Gamma \vdash I : \omega}{\Gamma \vdash call(I)\ \mathbf{ok}}$$

(SSEM - method call)

$$\frac{\langle I, (\sigma, h)\rangle \underset{inv}{\longrightarrow} \langle S, (\sigma', h')\rangle}{\langle call(I), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle seq(S, \texttt{pop}), (\sigma', h')\rangle}$$

(DSEM - method call)

$$\frac{\langle I, (\sigma, h)\rangle \underset{inv}{\longrightarrow} \langle \bot_{h'}\rangle}{\langle call(I), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \bot_{h'}\rangle}$$

(DSEM - method call: exception)

Invocations

An invocation is the process of executing the body of a constructor or method, and assigning the arguments to the formal parameters.

$$\begin{aligned} I \in \mathit{Invocation} ::= {} & \mathit{invoke}(i, M, \underline{E}) \\ \mid {} & \mathit{invoke}(C, M, \underline{E}) \\ \mid {} & \mathit{invoke}(K, \underline{E}) \end{aligned}$$

Non-Static Method Invocations. Non-static method invocations execute the body of a non-static method.

$$\mathit{invoke} : \mathit{Identifier} \times \mathit{Method} \times [\mathit{Expression}] \to \mathit{Invocation}$$

Non-static invocations are valid in the static semantics when the variable on which the method is called is a class type that contains the method that is called, and when the types of the formal parameters match that of the arguments.

$$\frac{\begin{array}{c} M = \mathit{method}(\texttt{false}, \omega, \_, \_, \underline{P}, S) \\ |\underline{P}| = |\underline{E}| \\ i : \mathit{ref}(C) \in \Gamma \\ C = \mathit{class}(\underline{M}, \_, \_) \\ M \in \underline{M} \\ \underline{\tau} = [\tau \mid \mathit{param}(\tau, \_) \in \underline{P}] \\ \forall j \in \{1, \ldots, |\underline{E}|\} : \Gamma \vdash \underline{E}_j : \underline{\tau}_j \\ \Gamma \vdash \texttt{this} : \mathit{ref}(C) \textbf{ ok in } S \\ \forall \mathit{param}(\tau_p, x_p) \in \underline{P} : \Gamma \vdash x_p : \tau_p \textbf{ ok in } S \end{array}}{\Gamma \vdash \mathit{invoke}(i, M, \underline{E}) : \omega} \quad \text{(SSEM - non-static invocation)}$$

$$\frac{\begin{array}{c} \sigma(i) \neq \texttt{null} \\ M = \mathit{method}(\_, \_, \_, \_, \underline{P}, S) \\ v_1 \neq \bot = \mathcal{E}[\![\underline{E}_1]\!](\sigma, h) \quad \ldots \quad v_n \neq \bot = \mathcal{E}[\![\underline{E}_n]\!](\sigma, h) \\ \underline{P} = [\mathit{param}(\_, x_1), \ldots, \mathit{param}(\_, x_n)] \\ \sigma' = \mathit{push}(\sigma)[\texttt{this} \mapsto i, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \end{array}}{\langle \mathit{invoke}(i, M, \underline{E}), (\sigma, h)\rangle \xrightarrow[\mathit{inv}]{} \langle S, (\sigma', h)\rangle} \quad \text{(DSEM - non-static invocation)}$$

$$\frac{\sigma(i) = \texttt{null}}{\langle \mathit{invoke}(i, M, \underline{E}), (\sigma, h)\rangle \xrightarrow[\mathit{inv}]{} \langle \bot_h\rangle} \quad \text{(DSEM - non-static invocation: null dereference)}$$

$$\frac{\exists E \in \underline{E} : \mathcal{E}[\![E]\!](\sigma, h) = \bot}{\langle \mathit{invoke}(i, M, \underline{E}), (\sigma, h)\rangle \xrightarrow[\mathit{inv}]{} \langle \bot_h\rangle} \quad \text{(DSEM - non-static invocation: evaluation exception)}$$

Static Method Invocations. Static method invocations execute the body of a static method.

$$\mathit{invoke} : \mathit{Class} \times \mathit{Method} \times [\mathit{Expression}] \to \mathit{Invocation}$$

Static invocations are valid in the static semantics when the class the method is called on exists and contains the method that is called. The types of the formal parameters must match that of the arguments.

$$\frac{\begin{array}{c} M = \mathit{method}(\texttt{true}, \omega, \_, \_, \underline{P}, S) \\ |\underline{P}| = |\underline{E}| \\ C = \mathit{class}(\underline{M}, \_, \_) \\ M \in \underline{M} \\ \underline{\tau} = [\tau \mid \mathit{param}(\tau, \_) \in \underline{P}] \\ \forall j \in \{1, \ldots, |\underline{E}|\} : \Gamma \vdash \underline{E}_j : \underline{\tau}_j \\ \forall \mathit{param}(\tau_p, x_p) \in \underline{P} : \Gamma \vdash x_p : \tau_p \textbf{ ok in } S \end{array}}{\Gamma \vdash \mathit{invoke}(C, M, \underline{E}) : \omega} \quad \text{(SSEM - static invocation)}$$

$$\frac{\begin{array}{c} M = \mathit{method}(\_, \_, \_, \_, \underline{P}, S) \\ v_1 \neq \bot = \mathcal{E}[\![\underline{E}_1]\!](\sigma, h) \quad \ldots \quad v_n \neq \bot = \mathcal{E}[\![\underline{E}_n]\!](\sigma, h) \\ \underline{P} = [\mathit{param}(\_, x_1), \ldots, \mathit{param}(\_, x_n)] \\ \sigma' = \mathit{push}(\sigma)[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \end{array}}{\langle \mathit{invoke}(C, M, \underline{E}), (\sigma, h) \rangle \underset{inv}{\longrightarrow} \langle S, (\sigma', h) \rangle} \quad \text{(DSEM - static invocation)}$$

$$\frac{\exists E \in \underline{E} : \mathcal{E}[\![E]\!](\sigma, h) = \bot}{\langle \mathit{invoke}(C, M, \underline{E}), (\sigma, h) \rangle \underset{inv}{\longrightarrow} \langle \bot_h \rangle} \quad \text{(DSEM - static invocation: evaluation exception)}$$

Constructor Invocations. Constructor invocations execute the body of a constructor after which it returns a newly allocated object.

$$\mathit{invoke} : \mathit{Constructor} \times [\mathit{Expression}] \to \mathit{Invocation}$$

Constructor invocations are valid in the static semantics when the class the constructor is called on exists and contains the constructor that is called. The types of the formal parameters must match that of the arguments.

$$\frac{\begin{array}{c} K = \mathit{constructor}(C, \_, \underline{P}, S) \\ |\underline{P}| = |\underline{E}| \\ \underline{\tau} = [\tau \mid \mathit{param}(\tau, \_) \in \underline{P}] \\ \forall j \in \{1, \ldots, |\underline{E}|\} : \Gamma \vdash \underline{E}_j : \underline{\tau}_j \\ \Gamma \vdash \texttt{this} : \mathit{ref}(C) \textbf{ ok in } S \\ \forall \mathit{param}(\tau_p, x_p) \in \underline{P} : \Gamma \vdash x_p : \tau_p \textbf{ ok in } S \end{array}}{\Gamma \vdash \mathit{invoke}(K, \underline{E}) : \mathit{type}(\mathit{ref}(C))} \quad \text{(SSEM - constructor invocation)}$$

$$\frac{\begin{array}{c} alloc(h) = ref_{fresh} \\ K = constructor(C, \_, \underline{P}, S) \\ C = class(\_, \_, \underline{F}) \\ \mathcal{O} = [x \mapsto default(\tau) \mid field(\tau, x) \in \underline{F}] \\ v_1 \neq \bot = \mathcal{E}[\![\underline{E}_1]\!](\sigma, h) \quad \ldots \quad v_n \neq \bot = \mathcal{E}[\![\underline{E}_n]\!](\sigma, h) \\ \underline{P} = [param(\_, x_1), \ldots, param(\_, x_n)] \\ \sigma' = push(\sigma)[\texttt{this} \mapsto ref_{fresh}, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \end{array}}{\langle invoke(K, \underline{E}), (\sigma, h)\rangle \xrightarrow[inv]{} \langle S, (\sigma', h[ref_{fresh} \mapsto \mathcal{O}])\rangle} \quad \text{(DSEM - constructor invocation)}$$

$$\frac{\exists E \in \underline{E} : \mathcal{E}[\![E]\!](\sigma, h) = \bot}{\langle invoke(K, \underline{E}), (\sigma, h)\rangle \xrightarrow[inv]{} \langle \bot_h \rangle} \quad \text{(DSEM - constructor invocation: evaluation exception)}$$

3.7.4 Skip Statements

A skip statement performs no operation.

$$skip : Statement$$

The dynamic semantics of skip statements is handled in the program execution. When a skip statement is single last statement of a thread, the thread is terminated. See Section 3.5 for more information.

3.7.5 Assert Statements

An assert statement instructs the verification back-end to verify whether the given expression is valid or not. If this is the case, the execution continues as normal, otherwise, the execution halts.

$$assert : Expression \rightarrow Statement$$

An assert statement is valid in the static semantics if the assertion is of type `bool`.

$$\frac{\Gamma \vdash E : \texttt{bool}}{\Gamma \vdash assert(E)\ \textbf{ok}} \quad \text{(SSEM - assert)}$$

$$\frac{}{\langle assert(E), (\sigma, h)\rangle \xrightarrow[stat]{} \langle \texttt{skip}, (\sigma, h)\rangle} \quad \text{(DSEM - assert)}$$

3.7.6 Assume Statements

An assume statement instructs the verification back-end to assume that the given expression holds in the rest of the execution.

$$assume : Expression \rightarrow Statement$$

An assume statement is valid in the static semantics if the assumption is of type `bool`.

$$\frac{\Gamma \vdash E : \texttt{bool}}{\Gamma \vdash \mathit{assume}(E)\ \mathbf{ok}} \quad \text{(SSEM - assume)}$$

$$\frac{}{\langle \mathit{assume}(E), (\sigma, h)\rangle \xrightarrow[stat]{} \langle \texttt{skip}, (\sigma, h)\rangle} \quad \text{(DSEM - assume)}$$

3.7.7 While Statements

A while statement executes the body of the loop as long as the guard evaluates to true.

$$\mathit{while} : \mathit{Expression} \times \mathit{Statement} \to \mathit{Statement}$$

A while statement is valid in the static semantics when the guard is of type `bool`.

$$\frac{\Gamma \vdash E : \texttt{bool}}{\Gamma \vdash \mathit{while}(E, S)\ \mathbf{ok}} \quad \text{(SSEM - while)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = \texttt{true}}{\langle \mathit{while}(E, S), (\sigma, h)\rangle \xrightarrow[stat]{} \langle \mathit{seq}(S, \mathit{while}(E, S)), (\sigma, h)\rangle} \quad \text{(DSEM - while: true case)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = \texttt{false}}{\langle \mathit{while}(E, S), (\sigma, h)\rangle \xrightarrow[stat]{} \langle \texttt{skip}, (\sigma, h)\rangle} \quad \text{(DSEM - while: false case)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = \bot}{\langle \mathit{while}(E, S), (\sigma, h)\rangle \xrightarrow[stat]{} \langle \bot_h\rangle} \quad \text{(DSEM - while: guard evaluation exception)}$$

3.7.8 If-Then-Else Statements

An if-then-else statement allows for conditional execution of statements.

$$\mathit{ite} : \mathit{Expression} \times \mathit{Statement} \times \mathit{Statement} \to \mathit{Statement}$$

A if-then-else statement is valid in the static semantics when the guard is of type `bool`.

$$\frac{\Gamma \vdash E : \texttt{bool}}{\Gamma \vdash \mathit{ite}(E, S_1, S_2)\ \mathbf{ok}} \quad \text{(SSEM - if-then-else)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = \texttt{true}}{\langle \mathit{ite}(E, S_1, S_2), (\sigma, h)\rangle \xrightarrow[stat]{} \langle S_1, (\sigma, h)\rangle} \quad \text{(DSEM - if-then-else: true case)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = \texttt{false}}{\langle \mathit{ite}(E, S_1, S_2), (\sigma, h)\rangle \xrightarrow[stat]{} \langle S_2, (\sigma, h)\rangle} \quad \text{(DSEM - if-then-else: false case)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = \bot}{\langle \mathit{ite}(E, S_1, S_2), (\sigma, h)\rangle \xrightarrow[stat]{} \langle \bot_{h'}\rangle} \quad \text{(DSEM - if-then-else: guard evaluation exception)}$$

3.7.9 Continue and Break Statements

The continue and break statements affect the control flow when placed inside a while loop. Both statements can only be placed within the body of a while loop.

Continue Statements. A continue statement transfers the control flow to the guard of the loop in which it is embedded.

$$\textit{continue} : \textit{Statement}$$

$$\frac{}{\Gamma \vdash \texttt{continue}\ \textbf{ok in}\ S\ \textbf{of}\ \textit{while}(E, S)} \quad \text{(SSEM - continue)}$$

Break Statements. A break statement transfers the control flow to the statement that comes after the loop in which it is embedded. For example, in the program `while(e) break; x := 1;`, the `break` statement transfers the control flow to the statement `x := 1;`.

$$\textit{break} : \textit{Statement}$$

$$\frac{}{\Gamma \vdash \texttt{break}\ \textbf{ok in}\ S\ \textbf{of}\ \textit{while}(E, S)} \quad \text{(SSEM - break)}$$

3.7.10 Return Statements

Return statements allow for the returning of values inside methods and constructors. A return statement transfers the control flow back to the statement that contained the method call. There are two kind of return statements: one that only terminates the current method, and one that terminates the current method and returns a value.

The first kind of return statement is defined as

$$\textit{return} : \textit{Statement}$$

The first kind of return statement is valid in the static semantics when the return type of the method call the return statement is contained in is of type `void`.

$$\frac{\omega = \texttt{void}}{\Gamma \vdash \texttt{return}\ \textbf{ok in}\ S\ \textbf{of}\ \textit{method}(\omega, i, \underline{P}, S)} \quad \text{(SSEM - return)}$$

$$\frac{}{\langle \texttt{return}, (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \texttt{skip}, (\sigma, h)\rangle} \quad \text{(DSEM - return)}$$

The second kind of return statement is defined as

$$\textit{return} : \textit{Expression} \rightarrow \textit{Statement}$$

The second kind of return statement is valid in the static semantics when the return type of the method call the return statement is contained in matches that of the supplied expression.

$$\frac{\omega = \mathit{type}(\tau) \quad \Gamma \vdash E : \tau}{\Gamma \vdash \mathit{return}(E) \textbf{ ok in } S \textbf{ of } \mathit{method}(\omega, i, \underline{P}, S)} \qquad \text{(SSEM - return)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = v \neq \bot}{\langle \mathit{return}(E), (\sigma, h)\rangle \xrightarrow[\mathit{stat}]{} \langle \texttt{skip}, (\sigma[\mathit{retval}' \mapsto v], h)\rangle} \qquad \text{(DSEM - return)}$$

$$\frac{\mathcal{E}[\![E]\!](\sigma, h) = \bot}{\langle \mathit{return}(E), (\sigma, h)\rangle \xrightarrow[\mathit{stat}]{} \langle \bot_h\rangle} \qquad \text{(DSEM - return: evaluation exception)}$$

3.7.11 Throw Statements

Throw statements are used to transfer the control flow to the exceptional state. When this exceptional state is encapsulated within a try-catch statement, the control flow is transferred to the corresponding catch block.

$$\mathit{throw} : \mathit{Statement}$$

$$\frac{}{\langle \texttt{throw}, (\sigma, h)\rangle \xrightarrow[\mathit{stat}]{} \langle \bot_h\rangle} \qquad \text{(DSEM - throw)}$$

3.7.12 Try Statements

Try statements allow for handling exceptional states.

$$\mathit{try} : \mathit{Statement} \times \mathit{Statement} \to \mathit{Statement}$$

$$\frac{\langle S_1, (\sigma, h)\rangle \xrightarrow[\mathit{stat}]{} \langle S_1', (\sigma', h')\rangle \quad S_1' \neq \texttt{skip}}{\langle \mathit{try}(S_1, S_2), (\sigma, h)\rangle \xrightarrow[\mathit{stat}]{} \langle \mathit{try}(S_1', S_2), (\sigma', h')\rangle} \qquad \text{(DSEM - try: normal execution)}$$

$$\frac{\langle S_1, (\sigma, h)\rangle \xrightarrow[\mathit{stat}]{} \langle \texttt{skip}, (\sigma', h')\rangle}{\langle \mathit{try}(S_1, S_2), (\sigma, h)\rangle \xrightarrow[\mathit{stat}]{} \langle \texttt{skip}, (\sigma', h')\rangle} \qquad \text{(DSEM - try: finished execution)}$$

$$\frac{\langle S_1, (\sigma, h)\rangle \xrightarrow[\mathit{stat}]{} \langle \bot_{h'}\rangle}{\langle \mathit{try}(S_1, S_2), (\sigma, h)\rangle \xrightarrow[\mathit{stat}]{} \langle S_2, (\sigma, h')\rangle} \qquad \text{(DSEM - try: exceptional execution)}$$

3.7.13 Block Statements

Block statements allow for the grouping of statements and introducing new scopes. Variables declared in this new scope are no longer available when the control flow is transferred to the statement that follows the block statement.

$$block : Statement \rightarrow Statement$$

A block statement is always valid in the static semantics.

$$\frac{}{\Gamma \vdash block(S)\ \mathbf{ok}} \quad \text{(SSEM - block)}$$

$$\frac{\sigma' = push(\sigma)}{\langle block(S), (\sigma, h)\rangle \xrightarrow[stat]{} \langle seq(S, \texttt{pop}), (\sigma', h)\rangle} \quad \text{(DSEM - block)}$$

3.7.14 Lock- and Unlock Statements

Lock and unlock statements allow for locking and unlocking a reference to the heap. When another thread tries to lock this reference, this thread waits until the reference is freed. This type of lock is regularly named a monitor.

$$lock : Identifier \rightarrow Statement$$

$$unlock : Identifier \rightarrow Statement$$

The lock and unlock statements are valid in the static semantics when the variable they aim to lock is a reference type.

$$\frac{i : \tau \in \Gamma \quad \tau \prec REF}{\Gamma \vdash lock(i)\ \mathbf{ok}} \quad \text{(SSEM - lock)}$$

$$\frac{i : \tau \in \Gamma \quad \tau \prec REF}{\Gamma \vdash unlock(i)\ \mathbf{ok}} \quad \text{(SSEM - unlock)}$$

The dynamic semantics of lock- and unlock statements is handled in the program execution. See Section 3.5 for more information.

3.7.15 Join Statements

A join statement lets the current thread wait if and until all child threads have terminated. The dynamic semantics of the join statement is handled in the program execution. See Section 3.5 for more information.

A join statement is always valid in the static semantics.

$$\frac{}{\Gamma \vdash \texttt{join}\ \mathbf{ok}} \quad \text{(SSEM - join)}$$

3.7.16 Fork Statements

A fork statement spawns a new thread starting at the given method invocation.

$$\mathit{fork} : \mathit{Invocation} \to \mathit{Statement}$$

$$\frac{\Gamma \vdash I : \omega}{\Gamma \vdash \mathit{fork}(I)\ \mathbf{ok}} \qquad \text{(SSEM - fork)}$$

The dynamic semantics of fork statements is handled in the program execution. See Section 3.5 for more information.

3.7.17 Sequence Statements

Sequence statements are implicit statements which are used to define two statements one executed after the other.

$$\mathit{seq} : \mathit{Statement} \times \mathit{Statement} \to \mathit{Statement}$$

$$\frac{S_1 = \mathit{declare}(\tau, i) \quad \Gamma \vdash i : \tau\ \mathbf{ok}\ \mathbf{in}\ S_2}{\Gamma \vdash \mathit{seq}(S_1, S_2)\ \mathbf{ok}} \qquad \text{(SSEM - sequence: declaration)}$$

$$\frac{S_1 \neq \mathit{declare}(\tau, i)}{\Gamma \vdash \mathit{seq}(S_1, S_2)\ \mathbf{ok}} \qquad \text{(SSEM - sequence: non-declaration)}$$

$$\frac{\langle S_1, (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle S_1', (\sigma', h')\rangle \quad S_1' \neq \texttt{skip}}{\langle \mathit{seq}(S_1, S_2), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \mathit{seq}(S_1', S_2), (\sigma', h')\rangle} \qquad \text{(DSEM - sequence: statement)}$$

$$\frac{\langle S_1, (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \texttt{skip}, (\sigma', h')\rangle}{\langle \mathit{seq}(S_1, S_2), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle S_2, (\sigma', h')\rangle} \qquad \text{(DSEM - sequence: skip statement)}$$

$$\frac{\langle S_1, (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \bot_h\rangle}{\langle \mathit{seq}(S_1, S_2), (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \bot_h\rangle} \qquad \text{(DSEM - sequence: exception)}$$

3.7.18 Pop Statements

Pop statements are implicit statements that pop the top-most stack frame from the stack.

$$\mathit{pop} : \mathit{Statement}$$

A pop statement is always valid in the static semantics.

$$\frac{}{\Gamma \vdash \texttt{pop}\ \mathbf{ok}} \qquad \text{(SSEM - pop)}$$

$$\frac{\sigma' = \mathit{pop}(\sigma)[\mathit{retval}' \mapsto \sigma(\mathit{retval}')]}{\langle \texttt{pop}, (\sigma, h)\rangle \underset{stat}{\longrightarrow} \langle \texttt{skip}, (\sigma', h)\rangle} \qquad \text{(DSEM - pop)}$$

3.8 Expressions

An expression is a sequence of operators and operands which result in a value when evaluated. The complete list of expressions in the abstract syntax of OOX is defined as

$$
\begin{array}{lll}
E \in \mathit{Expression} & ::= & \mathit{lit}(n) \mid \mathit{lit}(z) \mid \mathit{lit}(r) \mid \mathit{lit}(b) \mid \mathit{lit}(s) \mid \mathit{lit}(c) \\
& \mid & \mathit{ref}_n \\
& \mid & \texttt{null} \\
& \mid & \mathit{var}(i) \\
& \mid & \mathit{unop}(E, \oplus) \\
& \mid & \mathit{binop}(E_1, \otimes, E_2) \\
& \mid & \mathit{sizeof}(i) \\
& \mid & \mathit{ite}(E_1, E_2, E_3) \\
& \mid & \mathit{forall}(i_1, i_2, i_3, E) \\
& \mid & \mathit{exists}(i_1, i_2, i_3, E) \\
n \in \mathit{Nat} & ::= & \mathbb{N}_0 \\
z \in \mathit{Int} & ::= & \mathbb{Z} \\
r \in \mathit{Real} & ::= & \mathbb{R} \\
b \in \mathit{Bool} & ::= & \texttt{true} \mid \texttt{false} \\
s \in \mathit{String} & & \\
c \in \mathit{Char} & & \\
\oplus \in \mathit{UnaryOperator} & ::= & \texttt{!} \mid \texttt{-} \\
\otimes \in \mathit{BinaryOperator} & ::= & \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{+} \mid \texttt{-} \\
& \mid & \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \\
& \mid & \texttt{!=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{==>}
\end{array}
$$

3.8.1 Literals

OOX defines seven different literals. Below is a complete list of the literals in OOX.

Unsigned Integer Literals. The unsigned integer literals are the constant values that can represent the numbers in $\mathbb{N}_0$ up to but not including 2^{32}.

$$\mathit{lit} : \mathit{Nat} \to \mathit{Expression}$$

$$\frac{}{\Gamma \vdash \mathit{lit}(n) : \texttt{uint}} \qquad \text{(SSEM - unsigned integer literals)}$$

Signed Integer Literals. The signed integer literals are the constant values that can represent the numbers in $\mathbb{Z}$ from -2^{31} up to but not including 2^{31}.

$$\mathit{lit} : \mathit{Int} \to \mathit{Expression}$$

$$\frac{}{\Gamma \vdash \mathit{lit}(z) : \texttt{int}} \quad \text{(SSEM - signed integer literals)}$$

Floating Point Literals. The floating point literals are the constant values that can represent a subset of the numbers in $\mathbb{R}$.

$$\mathit{lit} : \mathit{Real} \to \mathit{Expression}$$

$$\frac{}{\Gamma \vdash \mathit{lit}(r) : \texttt{float}} \quad \text{(SSEM - floating point literals)}$$

Boolean Literals. The boolean literals are the constant values that represent truth values. That is, they can be either `true` or `false`.

$$\mathit{lit} : \mathit{Bool} \to \mathit{Expression}$$

$$\frac{}{\Gamma \vdash \mathit{lit}(b) : \texttt{bool}} \quad \text{(SSEM - boolean literals)}$$

String Literals. The string literals are the constant values that can represent human readable text.

$$\mathit{lit} : \mathit{String} \to \mathit{Expression}$$

$$\frac{}{\Gamma \vdash \mathit{lit}(s) : \texttt{string}} \quad \text{(SSEM - string literals)}$$

Character Literals. The character literals are the constant values that can represent a single human readable character.

$$\mathit{lit} : \mathit{Char} \to \mathit{Expression}$$

$$\frac{}{\Gamma \vdash \mathit{lit}(c) : \texttt{char}} \quad \text{(SSEM - character literals)}$$

3.8.2 References

A reference, also named pointers in some languages, represents an abstract memory address on the heap.

$$\mathit{ref} : \mathit{Nat} \to \mathit{Expression}$$

$$\frac{\tau \prec \mathit{REF}}{\Gamma \vdash \mathit{ref}(n) : \tau} \quad \text{(SSEM - references)}$$

The Null Literal. The `null` literal is a special kind of reference representing a reference that points to nothing on the heap. In other words, an unallocated value.

$$\mathit{null} : \mathit{Expression}$$

$$\frac{\tau \prec \mathit{REF}}{\Gamma \vdash \texttt{null} : \tau} \quad \text{(SSEM - the null literal)}$$

3.8.3 Variable Access

Variable access allows for the reading of a variable on the stack. For example, the variable `x` is read in the expression `x + 1`.

$$\mathit{var} : \mathit{Identifier} \to \mathit{Expression}$$

$$\frac{i : \tau \in \Gamma}{\Gamma \vdash \mathit{var}(i) : \tau} \quad \text{(SSEM - variable access)}$$

3.8.4 Unary Operators

OOX defines two unary operators: numerical negation, written using the `-` operator, and boolean negation, written using the `!` operator.

$$\mathit{unop} : \mathit{Expression} \times \mathit{UnaryOperator} \to \mathit{Expression}$$

$$\frac{\Gamma \vdash E : \tau \quad \tau \prec \mathit{NUM}}{\Gamma \vdash \mathit{unop}(E, \texttt{-}) : \tau} \quad \text{(SSEM - numeric negation)}$$

$$\frac{\Gamma \vdash E : \texttt{bool}}{\Gamma \vdash \mathit{unop}(E, \texttt{!}) : \texttt{bool}} \quad \text{(SSEM - boolean negation)}$$

3.8.5 Binary Operators

OOX defines four categories of binary operators: arithmetic-, comparison-, equality- and boolean operators.

$$\mathit{binop} : \mathit{Expression} \times \mathit{BinaryOperator} \times \mathit{Expression} \to \mathit{Expression}$$

Arithmetic Operators. The arithmetic operators take two numerical operands and when evaluated result in a numerical value. The operators that are defined are: multiplication, division, remainder, addition and subtraction, written using `*`, `/`, `%`, `+` and `-` respectively.

$$\frac{\begin{array}{c}\odot \in \{\texttt{*}, \texttt{/}, \texttt{\%}, \texttt{+}, \texttt{-}\} \\ \Gamma \vdash E_1 : \tau_1 \quad \tau_1 \prec \mathit{NUM} \\ \Gamma \vdash E_2 : \tau_2 \quad \tau_2 \prec \mathit{NUM}\end{array}}{\Gamma \vdash \mathit{binop}(E_1, \odot, E_2) : \tau_1} \quad \text{(SSEM - arithmetic operators)}$$

Comparison Operators. The comparison operators take two numerical operands and when evaluated result in a boolean value. The comparison operators that are defined are: less than, less than or equal, greater than and greater than or equal, written using `<`, `<=`, `>` and `>=` respectively.

$$\frac{\begin{array}{c}\odot \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\} \\ \Gamma \vdash E_1 : \tau_1 \quad \tau_1 \prec \mathit{NUM} \\ \Gamma \vdash E_2 : \tau_2 \quad \tau_2 \prec \mathit{NUM}\end{array}}{\Gamma \vdash \mathit{binop}(E_1, \odot, E_2) : \texttt{bool}} \qquad \text{(SSEM - comparison operators)}$$

Equality Operators. The equality operators take two operands that are of equal type and when evaluated result in a boolean value. The equality operators that are defined are: equality, written using `==`, and inequality, written using `!=`.

$$\frac{\Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau \quad \odot \in \{\texttt{==}, \texttt{!=}\}}{\Gamma \vdash \mathit{binop}(E_1, \odot, E_2) : \texttt{bool}} \qquad \text{(SSEM - equality operators)}$$

Boolean Operators. The boolean take two boolean operands and when evaluated result in a boolean value. The boolean operators that are defined are: conjunction, disjunction and implication, written using `&&`, `||` and `==>` respectively.

$$\frac{\Gamma \vdash E_1 : \texttt{bool} \quad \Gamma \vdash E_2 : \texttt{bool} \quad \odot \in \{\texttt{\&\&}, \texttt{||}, \texttt{==>}\}}{\Gamma \vdash \mathit{binop}(E_1, \odot, E_2) : \texttt{bool}} \qquad \text{(SSEM - boolean operators)}$$

3.8.6 Sizeof Operator

OOX defines a sizeof operator that takes a variable of array type as an operand and results in an integer value. The sizeof operator is written using `#`.

$$\mathit{sizeof} : \mathit{Identifier} \rightarrow \mathit{Expression}$$

$$\frac{i : \mathit{array}(\tau) \in \Gamma}{\Gamma \vdash \mathit{sizeof}(i) : \texttt{int}} \qquad \text{(SSEM - sizeof operator)}$$

3.8.7 If-Then-Else Operator

OOX defines an if-then-else operator which results either the evaluated second or third expression, depending on the value of the first expression.

$$\mathit{ite} : \mathit{Expression} \times \mathit{Expression} \times \mathit{Expression} \rightarrow \mathit{Expression}$$

$$\frac{\Gamma \vdash E_1 : \texttt{bool} \quad \Gamma \vdash E_2 : \tau \quad \Gamma \vdash E_3 : \tau}{\Gamma \vdash \mathit{ite}(E_1, E_2, E_3) : \tau} \qquad \text{(SSEM - ite operator)}$$

3.8.8 Quantifiers

OOX defines the two quantifiers from first-order logic: forall and exists. These quantifiers can be used to quantify over the elements and indices of an array.

$$\mathit{forall} : \mathit{Identifier} \times \mathit{Identifier} \times \mathit{Identifier} \times \mathit{Expression} \to \mathit{Expression}$$

$$\frac{\begin{array}{c} i_3 : \mathit{array}(\tau) \in \Gamma \\ \Gamma \vdash i_1 : \tau, i_2 : \texttt{int}\ \textbf{ok in}\ E \\ \Gamma \vdash E : \texttt{bool} \end{array}}{\Gamma \vdash \mathit{forall}(i_1, i_2, i_3, E) : \texttt{bool}} \qquad \text{(SSEM - forall quantifiers)}$$

$$\mathit{exists} : \mathit{Identifier} \times \mathit{Identifier} \times \mathit{Identifier} \times \mathit{Expression} \to \mathit{Expression}$$

$$\frac{\begin{array}{c} i_3 : \mathit{array}(\tau) \in \Gamma \\ \Gamma \vdash i_1 : \tau, i_2 : \texttt{int}\ \textbf{ok in}\ E \\ \Gamma \vdash E : \texttt{bool} \end{array}}{\Gamma \vdash \mathit{exists}(i_1, i_2, i_3, E) : \texttt{bool}} \qquad \text{(SSEM - exists quantifiers)}$$

3.8.9 The Dynamic Semantics

The dynamic semantics of expressions are defined using a denotational semantics. Let $\mathcal{E}$ be the valuation function

$$\mathcal{E}[\![E]\!] : \mathit{Expression} \times (\mathit{Stack} \times \mathit{Heap}) \to \mathit{Value}$$

where

$$\mathcal{E}[\![\mathit{lit}(x)]\!](\sigma, h) = x$$

$$\mathcal{E}[\![\mathit{var}(i)]\!](\sigma, h) = \sigma(i)$$

$$\begin{aligned} \mathcal{E}[\![\mathit{unop}(E, \texttt{-})]\!](\sigma, h) &= -\mathcal{E}[\![E]\!](\sigma, h) \\ \mathcal{E}[\![\mathit{unop}(E, \texttt{!})]\!](\sigma, h) &= \neg\mathcal{E}[\![E]\!](\sigma, h) \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\![\mathit{binop}(E_1, \texttt{*}, E_2)]\!](\sigma, h) &= \mathcal{E}[\![E_1]\!](\sigma, h) \cdot \mathcal{E}[\![E_2]\!](\sigma, h) \\ \mathcal{E}[\![\mathit{binop}(E_1, \texttt{/}, E_2)]\!](\sigma) &= \begin{cases} \bot & \textbf{if } \mathcal{E}[\![E_2]\!](\sigma, h) = 0 \\ \mathcal{E}[\![E_1]\!](\sigma, h) / \mathcal{E}[\![E_2]\!](\sigma, h) & \text{otherwise} \end{cases} \\ \mathcal{E}[\![\mathit{binop}(E_1, \texttt{\%}, E_2)]\!](\sigma, h) &= \begin{cases} \bot & \textbf{if } \mathcal{E}[\![E_2]\!](\sigma, h) = 0 \\ \mathcal{E}[\![E_1]\!](\sigma, h)(\mathit{mod}\ \ \mathcal{E}[\![E_2]\!](\sigma), h) & \text{otherwise} \end{cases} \\ \mathcal{E}[\![\mathit{binop}(E_1, \texttt{+}, E_2)]\!](\sigma, h) &= \mathcal{E}[\![E_1]\!](\sigma, h) + \mathcal{E}[\![E_2]\!](\sigma, h) \\ \mathcal{E}[\![\mathit{binop}(E_1, \texttt{-}, E_2)]\!](\sigma, h) &= \mathcal{E}[\![E_1]\!](\sigma, h) - \mathcal{E}[\![E_2]\!](\sigma, h) \end{aligned}$$

$$\mathcal{E}[\![binop(E_1, \texttt{<}, E_2)]\!](\sigma, h) = \mathcal{E}[\![E_1]\!](\sigma, h) < \mathcal{E}[\![E_2]\!](\sigma, h)$$
$$\mathcal{E}[\![binop(E_1, \texttt{<=}, E_2)]\!](\sigma, h) = \mathcal{E}[\![E_1]\!](\sigma, h) \leq \mathcal{E}[\![E_2]\!](\sigma, h)$$
$$\mathcal{E}[\![binop(E_1, \texttt{>}, E_2)]\!](\sigma, h) = \mathcal{E}[\![E_1]\!](\sigma, h) > \mathcal{E}[\![E_2]\!](\sigma, h)$$
$$\mathcal{E}[\![binop(E_1, \texttt{>=}, E_2)]\!](\sigma, h) = \mathcal{E}[\![E_1]\!](\sigma, h) \geq \mathcal{E}[\![E_2]\!](\sigma, h)$$

$$\mathcal{E}[\![binop(E_1, \texttt{==}, E_2)]\!](\sigma, h) = \mathcal{E}[\![E_1]\!](\sigma, h) = \mathcal{E}[\![E_2]\!](\sigma, h)$$
$$\mathcal{E}[\![binop(E_1, \texttt{!=}, E_2)]\!](\sigma, h) = \mathcal{E}[\![E_1]\!](\sigma, h) \neq \mathcal{E}[\![E_2]\!](\sigma, h)$$

$$\mathcal{E}[\![binop(E_1, \texttt{\&\&}, E_2)]\!](\sigma, h) = \mathcal{E}[\![E_1]\!](\sigma, h) \wedge \mathcal{E}[\![E_2]\!](\sigma, h)$$
$$\mathcal{E}[\![binop(E_1, \texttt{||}, E_2)]\!](\sigma, h) = \mathcal{E}[\![E_1]\!](\sigma, h) \vee \mathcal{E}[\![E_2]\!](\sigma, h)$$
$$\mathcal{E}[\![binop(E_1, \texttt{==>}, E_2)]\!](\sigma) = \mathcal{E}[\![E_1]\!](\sigma, h) \Rightarrow \mathcal{E}[\![E_2]\!](\sigma, h)$$

$$\mathcal{E}[\![sizeof(i)]\!](\sigma, h) = \begin{cases} \bot & \text{if } \sigma(i) = \texttt{null} \\ size(h(\sigma(i))) & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![ite(E_1, E_2, E_3)]\!](\sigma, h) = \begin{cases} \mathcal{E}[\![E_2]\!](\sigma, h) & \textbf{if } \mathcal{E}[\![E_1]\!](\sigma, h) = \texttt{true} \\ \mathcal{E}[\![E_3]\!](\sigma, h) & \textbf{if } \mathcal{E}[\![E_1]\!](\sigma, h) = \texttt{false} \end{cases}$$

$$\mathcal{E}[\![forall(i_1, i_2, i_3, E)]\!](\sigma, h) = \begin{cases} \bot & \textbf{if } \sigma(i_3) = \texttt{null} \\ \wedge_{j=0}^{n-1} \mathcal{E}[\![E]\!](\sigma[i_1 \mapsto v_j, i_2 \mapsto j], h) & \textbf{otherwise} \end{cases}$$
$$\mathcal{E}[\![exists(i_1, i_2, i_3, E)]\!](\sigma, h) = \begin{cases} \bot & \textbf{if } \sigma(i_3) = \texttt{null} \\ \vee_{j=0}^{n-1} \mathcal{E}[\![E]\!](\sigma[i_1 \mapsto v_j, i_2 \mapsto j], h) & \textbf{otherwise} \end{cases}$$

where $h(\sigma(i_3)) = \mathcal{O} = [elem_0 \mapsto v_0, \ldots, elem_n \mapsto v_n)$

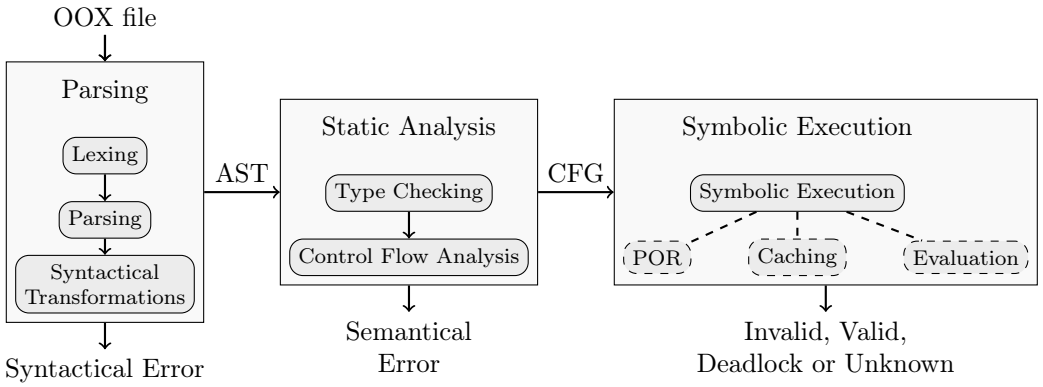
Chapter 4

Symbolic Execution of OOX Programs

Symbolic Execution (SE) was introduced by James C. King [25] as a means of executing a program where some variables are left symbolically, in contrast to concrete values. Symbolic execution has multiple use cases, but in the context of this thesis, symbolic execution is used to verify program correctness. Symbol execution is driven by the Symbolic Execution Engine (SEE), which operates on the Symbolic State (SS).

The Symbolic Execution Engine consists of three main phases, as presented in Figure 4.1. These three phases are: (1) parsing; (2) static analysis; and (3) symbolic execution. In the first two phases, the OOX program is preprocessed to be handled safely during the symbolic execution. That is, the exceptions are inserted as explicit if-then-else statements, the guards if-then-else statements and loops are inserted as explicit `assume` statements, and the program is verified to be free of syntactical- and (static) semantical errors. The SEE terminates with either a syntactical- or (static) semantical error, or with a verification result, which can be either that the program is considered invalid, valid, will result in a deadlock or that it is unknown.

Figure 4.1: The global architecture of the Symbolic Execution Engine.



Example 2 (A quick overview of the SEE.)**.** Suppose we have the method as presented in Listing 4.1 and suppose that this method is contained in the class `Math`. This function takes two `int` arguments, `x` and `y`, and returns the maximum of the two. Its specification describes that the return value must be greater than both `x` and `y`, and that te method will not terminate with an exception.

The first step of the SEE is to parse the source code into the AST and apply the syntactical transformations that yield the method as presented in Listing 4.2. See Section 4.1 for more details on the syntactical transformations that are applied.

Listing 4.1: The original max method.

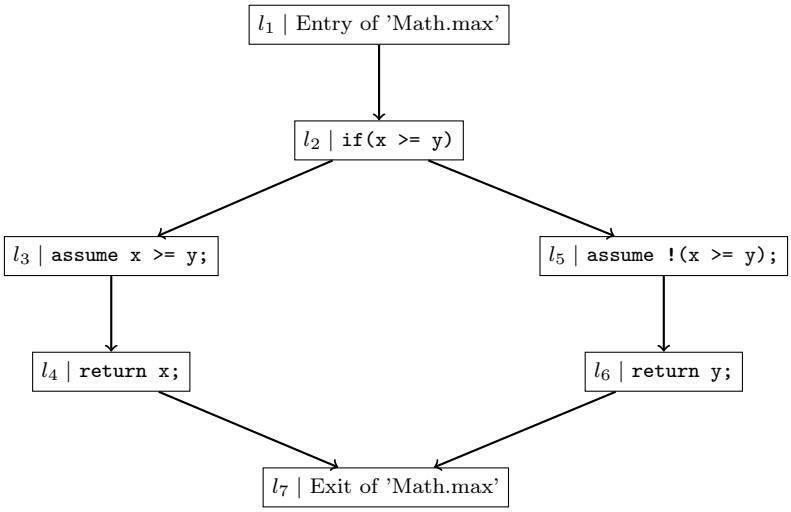
```
1  static int max(int x, int y)
2      ensures(retval >= x && retval >= y)
3      exceptional(false)
4  {
5      if (x >= y) {
6          return x;
7      } else {
8          return y;
9      }
10 }
```

Listing 4.2: The max method with the syntactical transformations applied.

```
1  static int max(int x, int y)
2      ensures(retval >= x && retval >= y)
3      exceptional(false)
4  {
5      if (x >= y) {
6          assume x >= y;
7          return x;
8      } else {
9          assume !(x >= y);
10         return y;
11     }
12 }
```

The next step is to verify the static semantics of the transformed program and if this succeeds, construct CFG from the AST. This yields the CFG as presented in Figure 4.2.

Figure 4.2: The CFG of the max method.



The third step is to execute the SEE algorithm, as presented in Section 4.3. The SEE algorithm, among other functionality, explores the search space, updates the Symbolic State and queries Z3 when necessary.

The search space consists of two program paths

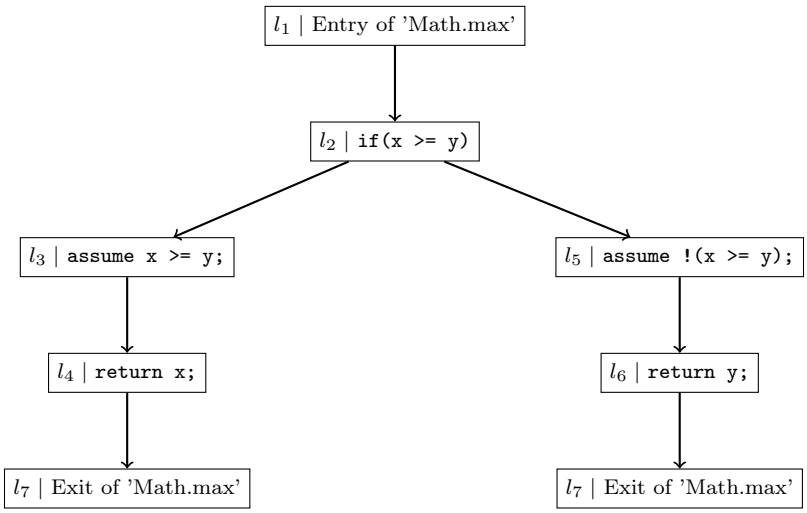
$$p_1 = l_1; l_2; l_3; l_4; l_7; \qquad p_2 = l_1; l_2; l_5; l_6; l_7;$$

as presented in Figure 4.3. We begin with an initial state $state_0$ in which only the main thread, $\mathcal{T}_1$, exists. The local variable of the call stack of $\mathcal{T}_1$ consists of two variables: `x` and `y`, which map to the symbolic variables α and β respectively. Let us exemplify the case in which the SEE verifies program path p_1. It starts with the actions corresponding to vertex l_1 and l_2, which are treated as `skip` statements in this particular example. It becomes interesting at the action corresponding to vertex l_3. This statement modifies the path constraints π from $\emptyset$ to the evaluated expression $\alpha \geq \beta$. The action at vertex l_4 assigns α to the variable *retval*. Finally the action at vertex l_7 will trigger the verification of the `ensures` specification. The `ensures` specification will combine the path constraints with the assertion to the formula

$$\alpha \geq \beta \wedge \neg(\alpha \geq \alpha \wedge \alpha \geq \beta)$$

The SEE queries Z3 to check whether this formula is satisfiable. When the formula is satisfiable, there exists a counter example and the specification does not hold. The same process is executed for program path p_2. Both these program paths yield an unsatisfiable specification which implies that the program satisfies its specification.

Figure 4.3: The search space of the max method.



□

4.1 Parsing

During the first phase of the Symbolic Execution Engine, the input file is lexed and parsed according to the concrete syntax, as defined in Appendix A. This will result into an AST, on which two syntactical transformations are applied: (1) every program point that can throw an exception, which is not an explicit `throw` statement, will be guarded with an if-then-else

statement; and (2) the guards of each if-then-else and while statement will be inserted explicitly as `assume` statements at their appropriate positions.

The first kind of transformation is that every program point that can throw an exception and is not an `throw` statement is embedded within an if-then-else statement. The guard of this if-then-else statement represents the condition for the exception to be thrown, the true branch contains a `throw` statement and the false branch contains the original statement.

Example 3. Suppose we have the statement `x := a[e];`. This statement throws an exception if either `a == null` or if the index is outside the bounds of the array. The statement is transformed into

```
if (a == null || e < 0 || e >= #a) throw; else x := a[e];
```

□

The condition for some exception to occur in statement S is defined by the function *exceptions*. The statement S is transformed using $transform : Statement \rightarrow Statement$. The *transform* function is applied to every statement in which an exception might occur. That is, every statement S in the program is transformed using:

$$transform(S) = \begin{cases} S & \textbf{if } exceptional(S) = \emptyset \\ ite(E_1 \,||\, \ldots \,||\, E_n, \texttt{throw}, S) & \textbf{if } exceptional(S) = \{E_1, \ldots, E_n\} \end{cases}$$

Where the *exceptions* function is defined as

$$\begin{aligned} exceptions(assign(t, v)) &= exceptions(t) \cup exceptions(v) \\ exceptions(call(I)) &= exceptions(I) \\ exceptions(assert(E)) &= exceptions(E) \\ exceptions(assume(E)) &= exceptions(E) \\ exceptions(while(E, S)) &= exceptions(E) \\ exceptions(ite(E, S_1, S_2)) &= exceptions(E) \\ exceptions(return(E)) &= exceptions(E) \\ exceptions(lock(i)) &= \{i == \texttt{null}\} \\ exceptions(unlock(i)) &= \{i == \texttt{null}\} \\ exceptions(fork(I)) &= exceptions(I) \\ exceptions(S) &= \emptyset \end{aligned}$$

$$\begin{aligned} exceptions(lhs_{var}(i)) &= \emptyset \\ exceptions(lhs_{field}(i, F)) &= \{i == \texttt{null}\} \\ exceptions(lhs_{elem}(i, E)) &= \{i == \texttt{null}, !(0 <= E < \#i)\} \cup exceptions(E) \end{aligned}$$

$$\begin{aligned} exceptions(rhs_{expr}(E)) &= exceptions(E) \\ exceptions(rhs_{field}(i, F)) &= \{i == \texttt{null}\} \\ exceptions(rhs_{call}(I)) &= exceptions(I) \\ exceptions(rhs_{elem}(i, E)) &= \{i == \texttt{null}, !(0 <= E < \#i)\} \cup exceptions(E) \\ exceptions(rhs_{array}(\tau, \underline{E})) &= \bigcup_{E \in \underline{E}} \{!(0 <= E)\} \cup exceptions(E) \end{aligned}$$

$$exceptions(invoke(i, M, \underline{E})) = \bigcup_{E \in \underline{E}} exceptions(E) \cup \{i == \texttt{null}\}$$
$$exceptions(invoke(C, M, \underline{E})) = \bigcup_{E \in \underline{E}} exceptions(E)$$
$$exceptions(invoke(K, \underline{E})) = \bigcup_{E \in \underline{E}} exceptions(E)$$

$$exceptions(E_1 \,/\, E_2) = \{E_2 == 0\} \cup exceptions(E_1) \cup exceptions(E_e)$$
$$exceptions(E_1 \,\%\, E_2) = \{E_2 == 0\} \cup exceptions(E_1) \cup exceptions(E_e)$$

The second kind of transformation is that for each if-then-else- and while statement in the program, the guard is inserted explicitly as an assume statement.

Example 4. Suppose we have the statement `while(e) { x := x + 1; }`. This statement will be transformed into the statements

```
while (e) { assume e; x := x + 1; } assume !e;
```

□

That is, every branching statement S in the program is transformed using

$$transform(ite(E, S_1, S_2)) = ite(E, seq(assume(E), S_1), seq(assume(!E), S_2))$$
$$transform(while(E, S)) = seq(while(E, seq(assume(E), S)), assume(!E))$$

4.2 Static Analysis

The static analysis consists of two parts: the verification of the static semantics and the control flow analysis. The verification of the static semantics includes type checking, as defined throughout Chapter 3. This type information is available during the symbolic execution and is, among other things, used to limit the effects of the path explosion problem arising from aliasing.

4.2.1 Control Flow Analysis

Control Flow Analysis (CFA) is a static analysis to determine the control flow of a program. Different kinds of control flow can be defined. For example: the intraprocedural control flow and the interprocedural control flow. The difference between the two being that the second captures the control flow of method calls to their respective methods. The CFA opted for is an intraprocedural procedural control flow. The invocations and their callbacks are managed in the SEE. The control flow is described using the Control Flow Graph (CFG).

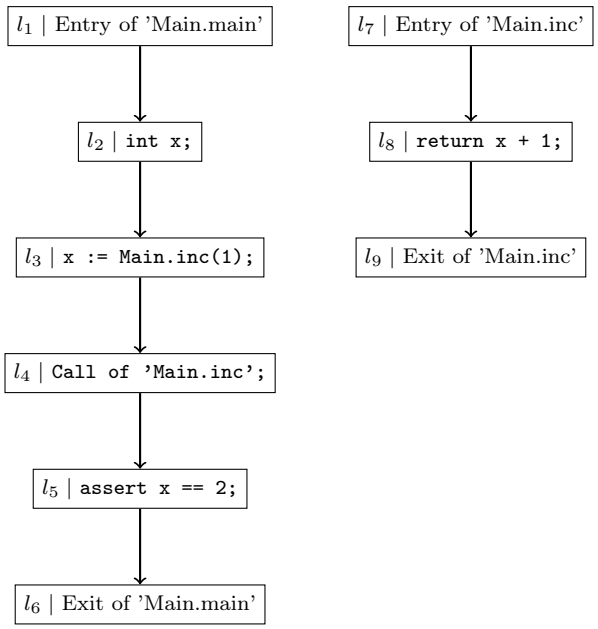
Definition 6 (Control Flow Graph)**.** The CFG is a directed graph $G = (V, E)$ where each node $v \in V$ represents a program point and each edge $(v, w) \in E$ represents the possible control flow between those program points. Let $action : V \to Action$ be a total function to retrieve the corresponding action of a vertex.

Example 5 (The CFG representation of methods and method calls.)**.** Suppose we have the program in Listing 4.3, which increments the value 1 via a method call. The CFG of the whole program is shown in Figure 4.4. The control flow between the different methods are disconnected. The Symbolic Execution Engine is responsible to find which method is called and it transfers the control flow to the correct method.

Listing 4.3: A `main` method with an `inc` function.

```
1  class Main {
2      static int main() {
3          int x;
4          x := Main.inc(1);
5          assert x == 2;
6      }
7
8      static int inc(int x) {
9          return x + 1;
10     }
11 }
```

Figure 4.4: The CFG of the program in Listing 4.3.



□

Each program point represents an executable position of the program, which is named an action.

Definition 7 (Action)**.** An action $a = action(v)$ of a vertex $v \in V$ is either: (1) a statement; (2) the entry point of a method or constructor; (3) the exit point of a method or constructor; (4) the call of a method or constructor; (5) the fork of a new thread; (6) the entry point of a try block; (7) the exit point of a try block; (8) the entry point of a catch block; (9) the exit point of a catch block; or (10) the exceptional state.

The CFG is constructed using the approach taken by Nielson et al. [29], adapted for the OOX language and extended with support for the control flow breaking statements `break`, `continue`, `return` and `throw`.

Let us first give an introduction to the approach taken by Nielson et al. Suppose that each node in the AST has an unique integer label l and suppose that each `try-catch` node has four unique integer (l_1, l_2, l_3, l_4).[1] Let these labels be ordered in increasing order from top to bottom and left to right, as appearing in the original source code. Then, three functions are defined: $\mathit{init} : \mathit{Statement} \to V$, $\mathit{final} : \mathit{Statement} \to \mathcal{P}(V)$ and $\mathit{flow} : \mathit{Statement} \to \mathcal{P}(E)$, where *init* defines the initial label of a statement, *final* defines the final labels of a statement and *flow* defines the control flow of a statement.

Let the *init* function be defined as the entry point of each statement, that is

$$
\begin{aligned}
\mathit{init}(\mathit{declare}(\tau, i)^l) &= l & \mathit{init}(\texttt{return}^l) &= l \\
\mathit{init}(\mathit{assign}(t, v)^l) &= l & \mathit{init}(\mathit{return}(E)^l) &= l \\
\mathit{init}(\mathit{call}(I)^l) &= l & \mathit{init}(\texttt{throw}^l) &= l \\
\mathit{init}(\texttt{skip}^l) &= l & \mathit{init}(\mathit{try}(S_1, S_2)^{(l_1,l_2,l_3,l_4)}) &= l_1 \\
\mathit{init}(\mathit{assert}(E)^l) &= l & \mathit{init}(\mathit{block}(S)^l) &= \mathit{init}(S) \\
\mathit{init}(\mathit{assume}(E)^l) &= l & \mathit{init}(\mathit{lock}(i)^l) &= l \\
\mathit{init}(\mathit{while}(E, S)^l) &= l & \mathit{init}(\mathit{unlock}(i)^l) &= l \\
\mathit{init}(\mathit{ite}(E, S_1, S_2)^l) &= l & \mathit{init}(\texttt{join}^l) &= l \\
\mathit{init}(\texttt{continue}^l) &= l & \mathit{init}(\mathit{fork}(I)^l) &= l \\
\mathit{init}(\texttt{break}^l) &= l & \mathit{init}(\mathit{seq}(S_1, S_2)^l) &= \mathit{init}(S_1)
\end{aligned}
$$

The approach is extended with the $\mathit{fallthrough} : \mathit{Statement} \to \mathcal{P}(\mathit{Statement})$ function which defines the control flow breaking statements contained in the statement. The function generates a singleton for the control flow breaking statements

$$
\begin{aligned}
\mathit{fallthrough}(\texttt{break}^l) &= \{\texttt{break}^l\} \\
\mathit{fallthrough}(\texttt{continue}^l) &= \{\texttt{continue}^l\} \\
\mathit{fallthrough}(\texttt{return}^l) &= \{\texttt{return}^l\} \\
\mathit{fallthrough}(\mathit{return}(E)^l) &= \{\mathit{return}(E)^l\}
\end{aligned}
$$

which are removed in their respective targets

$$
\begin{aligned}
\mathit{fallthrough}(\mathit{while}(E, S)^l) &= \{S' \mid S' \neq \texttt{continue}^{l'} \in \mathit{fallthrough}(S)\} \\
\mathit{fallthrough}(\mathit{seq}(\mathit{while}(E, S)^l, S_2)) &= \{S' \mid S' \neq \texttt{break}^{l'} \in \mathit{fallthrough}(S)\} \cup \mathit{fallthrough}(S_2)
\end{aligned}
$$

Let the *final* function be defined as the exit point of each statement. It is defined as the singleton

[1] `try-catch` statements are a special case. They do not appear explicitly in the CFG, but four nodes are inserted: the entry and exit of both the try and catch block.

label for the statements that do not contain nested statements.

$$\begin{aligned}
\mathit{final}(\mathit{declare}(\tau, i)^l) &= \{l\} & \mathit{final}(\mathit{assert}(E)^l) &= \{l\} \\
\mathit{final}(\mathit{assign}(t, \mathit{rhs}_{\mathit{call}}(I^{l'})^l) &= \{l'\} & \mathit{final}(\mathit{assume}(E)^l) &= \{l\} \\
\mathit{final}(\mathit{assign}(t, v)^l) &= \{l\} & \mathit{final}(\mathit{lock}(i)^l) &= \{l\} \\
\mathit{final}(\mathit{call}(I^{l'})^l) &= \{l'\} & \mathit{final}(\mathit{unlock}(i)^l) &= \{l\} \\
\mathit{final}(\mathit{fork}(I^{l'})^l) &= \{l'\} & \mathit{final}(\texttt{join}^l) &= \{l\} \\
\mathit{final}(\texttt{skip}^l) &= \{l\}
\end{aligned}$$

For the control flow breaking statements, the *final* function is defined as the empty set. Their induced control flow is captured in their respective targets.

$$\begin{aligned}
\mathit{final}(\texttt{continue}^l) &= \emptyset & \mathit{final}(\mathit{return}(E)^l) &= \emptyset \\
\mathit{final}(\texttt{break}^l) &= \emptyset & \mathit{final}(\texttt{throw}^l) &= \emptyset \\
\mathit{final}(\texttt{return}^l) &= \emptyset
\end{aligned}$$

The *final* function for statements that contain statements themselves is defined as

$$\begin{aligned}
\mathit{final}(\mathit{while}(E, S)^l) &= \{l\} \cup \{l' \mid \texttt{break}^{l'} \in \mathit{fallthrough}(S)\} \\
\mathit{final}(\mathit{ite}(E, S_1, S_2)^l) &= \mathit{final}(S_1) \cup \mathit{final}(S_2) \\
\mathit{final}(\mathit{try}(S_1, S_2)^{(l_1,l_2,l_3,l_4)}) &= \{l_2, l_4\} \\
\mathit{final}(\mathit{block}(S)^l) &= \mathit{final}(S)
\end{aligned}$$

Finally, for the `seq` statements, the *final* function is defined as

$$\begin{aligned}
\mathit{final}(\mathit{seq}(\texttt{continue}, S_2)^l) &= \emptyset & \mathit{final}(\mathit{seq}(\mathit{return}(E), S_2)^l)) &= \emptyset \\
\mathit{final}(\mathit{seq}(\texttt{break}, S_2)^l) &= \emptyset & \mathit{final}(\mathit{seq}(\texttt{throw}^{l'}, S_2)^l) &= \emptyset \\
\mathit{final}(\mathit{seq}(\texttt{return}, S_2)^l) &= \emptyset & \mathit{final}(\mathit{seq}(S_1, S_2)^l) &= \mathit{final}(S_2)
\end{aligned}$$

Let the *flow* function be defined as the control flow induced by the statements. For statements that do not contain nested statements it is defined as

$$\begin{aligned}
\mathit{flow}(\mathit{declare}(\tau, i)^l) &= \emptyset & \mathit{flow}(\texttt{break}^l) &= \emptyset \\
\mathit{flow}(\mathit{assign}(t, \mathit{rhs}_{\mathit{call}}(I^{l'})^l) &= \{(l, l')\} & \mathit{flow}(\texttt{return}^l) &= \emptyset \\
\mathit{flow}(\mathit{assign}(t, v)^l) &= \emptyset & \mathit{flow}(\mathit{return}(E)^l) &= \emptyset \\
\mathit{flow}(\mathit{call}(I^{l'})^l) &= \{(l, l')\} & \mathit{flow}(\texttt{throw}^l) &= \{(l, l_\perp)\} \\
\mathit{flow}(\texttt{skip}^l) &= \emptyset & \mathit{flow}(\mathit{lock}(i)^l) &= \emptyset \\
\mathit{flow}(\mathit{assert}(E)^l) &= \emptyset & \mathit{flow}(\mathit{unlock}(i)^l) &= \emptyset \\
\mathit{flow}(\mathit{assume}(E)^l) &= \emptyset & \mathit{flow}(\texttt{join}^l) &= \emptyset \\
\mathit{flow}(\texttt{continue}^l) &= \emptyset & \mathit{flow}(\mathit{fork}(I^{l'})^l) &= \{(l, l')\}
\end{aligned}$$

where $l_\perp$ is a special label that denotes the label of the exceptional state.

For the statements that contain nested statements, the *flow* function is defined as

$$\begin{aligned}
\mathit{flow}(\mathit{while}(E,S)^l) &= \mathit{flow}(S) \cup \{(l, \mathit{init}(S))\} \\
&\cup \{(l_f, l) \mid l_f \in \mathit{final}(S)\} \\
&\cup \{(l', l) \mid \texttt{continue}^{l'} \in \mathit{fallthrough}(S)\} \\
\mathit{flow}(\mathit{ite}(E,S_1,S_2)^l) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(l, \mathit{init}(S_1)), (l, \mathit{init}(S_2))\} \\
\mathit{flow}(\mathit{try}(S_1,S_2)^{(l_1,l_2,l_3,l_4)}) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \\
&\cup \{(l_1, \mathit{init}(S_1)), (l_3, \mathit{init}(S_2))\} \\
&\cup \{(l_f, l_2) \mid l_f \in \mathit{final}(S_1)\} \\
&\cup \{(l_f, l_4) \mid l_f \in \mathit{final}(S_2)\} \\
\mathit{flow}(\mathit{block}(S)^l) &= \mathit{flow}(S)
\end{aligned}$$

Finally, for the `seq` statements, the *flow* function is defined as

$$\begin{aligned}
\mathit{flow}(\mathit{seq}(\texttt{continue}, S_2)^l) &= \emptyset \\
\mathit{flow}(\mathit{seq}(\texttt{break}, S_2)^l) &= \emptyset \\
\mathit{flow}(\mathit{seq}(\texttt{return}, S_2)^l) &= \emptyset \\
\mathit{flow}(\mathit{seq}(\mathit{return}(E), S_2)^l) &= \emptyset \\
\mathit{flow}(\mathit{seq}(\texttt{throw}^{l'}, S_2)^l) &= \mathit{flow}(\texttt{throw}^{l'}) \\
\mathit{flow}(\mathit{seq}(S_1,S_2)^l) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(l_f, \mathit{init}(S_2)) \mid l_f \in \mathit{final}(S_1)\}
\end{aligned}$$

Let flow_U define the control flow of a compilation unit, let flow_C define the control flow of a class, and let flow_M and flow_K define the control flow of a method and constructor respectively. We then define their control flow as

$$\mathit{flow}_U(\mathit{program}(\underline{C})) = \bigcup_{C \in \underline{C}} \mathit{flow}_C(C)$$

$$\mathit{flow}_C(\mathit{class}(\underline{M}, \underline{K}, \underline{F})) = \bigcup_{M \in \underline{M}} \mathit{flow}_M(M) \cup \bigcup_{K \in \underline{K}} \mathit{flow}_K(K))$$

$$\begin{aligned}
\mathit{flow}_M(\mathit{method}(b, \omega, i, \varphi, \underline{P}, S)^{(l_i,l_f)}) &= \mathit{flow}(S) \cup \{(l_i, \mathit{init}(S))\} \\
&\cup \{(l, l_f) \mid l \in \mathit{final}(S)\} \\
&\cup \{(l, l_f) \mid S^l \in \mathit{fallthrough}(S)\}
\end{aligned}$$

$$\begin{aligned}
\mathit{flow}_K(\mathit{constructor}(C, \varphi, \underline{P}, S)^{(l_i,l_f)}) &= \mathit{flow}(S) \cup \{(l_i, \mathit{init}(S))\} \\
&\cup \{(l, l_f) \mid l \in \mathit{final}(S)\} \\
&\cup \{(l, l_f) \mid S^l \in \mathit{fallthrough}(S)\}
\end{aligned}$$

The complete Control Flow Graph of the program $U = \mathit{program}(\underline{C})$ is then defined in terms of the flow $E = \mathit{flow}_U(\mathit{program}(\underline{C}))$

$$\mathit{CFG} = (\cup_{(v,w) \in E}\{v, w\}, E)$$

4.3 The Symbolic Execution Engine

The Symbolic Execution Engine is the main process that drives the verification. There are a wide variety of design choices [3]. For example: is the exploration forward [20, 27], from the starting point of the program towards the end point of the program, or backward [22], from the end point of the program towards the starting point of the program. Another major design choice is in what way to explore the search space. Common considerations are breadth-first search, depth-first search, a heuristic, or a combination of these.

Our Symbolic Execution Engine uses a forward, depth-first path exploration approach. The logic to handle references to objects on the heap is handled in the SEE and the formula solving is transferred to Z3 [13], a SMT solver. This allows for greater flexibility of handling references, like lazy reference initialization and symbolic reference concretization. Four main optimization techniques are implemented: (1) lazy symbolic reference initialization; (2) formula caching; (3) expression evaluation; and (4) partial-order reduction. Each of these optimizations will be explained in detail in the coming sections.

The main structure of the Symbolic Execution Engine is the Symbolic State, which is defined as:

Definition 8 (The Symbolic State)**.** The SS is a 6-tuple $state = (\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$ where

- $\mathcal{T}$ is the set of threads;
- h is the heap, a mapping from references object structures. An object structure is a mapping from fields (or indices) to expressions;
- π is the set of path constraints that must be satisfiable for the current program path to be reachable;
- $\mathcal{A}$ is the mapping from symbolic references to their possible concrete references;
- $\mathcal{L}$ is the mapping from reference to thread id, representing the references that are currently locked by which threads; and
- $\mathcal{I}$ is the set of interleaving constraints.

4.3.1 The Threads

The basis of the executable program are the threads. Each thread is a three-tuple $\mathcal{T}_{tid} = (pc, \sigma, \eta)$ consisting of:

- a program counter $pc \in V$;
- the call stack σ consisting of stack frames; and
- the exception handler stack η.

A thread is disabled if it is waiting for a lock to be freed, or the current action is a `join` statement and there exists a child thread of this thread. A thread is enabled if it is not disabled. Let $parent(tid)$ be a function that returns the thread id of the parent of the thread with thread id tid.

The Call Stack. The call stack consists of the data necessary to keep track of the method calls that are made. Tt contains the local variables, each in their respective stack frame. Each method call is represented as a stack frame on the call stack. The call stack is defined as a sequence of stack frames, and is denoted by σ. New stack frames can be pushed and popped from the stack using

$$push_{stack}(\sigma, pc, t) = [(pc, t, []), \sigma_1, \ldots, \sigma_n] \textbf{ where } \sigma = [\sigma_1, \ldots, \sigma_n]$$

$$pop_{stack}(\sigma) = (\sigma_1, \ldots, \sigma_n] \textbf{ where } \sigma = [\sigma_1, \ldots, \sigma_n]$$

Each stack frame is a triple $(pc, t, \mathcal{M})$ where pc represents the program counter to return to when the method has finished its execution, t is the left-hand side the return value of the method will be assigned to and $\mathcal{M}$ are the local variables.

The Exception Handler Stack. The exception handler stack manages the data necessary to know whether or not the current statement is within a try block and where to return to if an exception occurs. The exception handler stack is defined as a sequence of 2-tuples, denoted by η. The first element of the tuple defines the program point of the catch entry that belongs to the current try block. The second element of the tuple defines the current depth of the call stack relative to when the try block point was entered. Exception handlers can be inserted and removed using

$$insert_{handler}(\eta, (pc, n)) = [(pc, n), \eta_1, \ldots, \eta_n] \textbf{ where } \eta = [\eta_1, \ldots, \eta_n]$$

$$remove_{handler}(\eta) = (\eta_1, \ldots, \eta_n] \textbf{ where } \eta = [\eta_1, \ldots, \eta_n]$$

4.3.2 The Memory

The memory consists of two distinct parts: the local variables, contained in the stack frames, and the heap. The values of the local variables are represented using the expression language of the OOX language. The objects on the heap are represented via an object structure, which is defined as a mapping from fields or indices to expressions.

The main difference between the expression language of OOX and the expression language of the SEE is that literals and references can be symbolic. A symbolic reference or value can represent multiple concrete values, possibly constrained by the path constraints π. The symbolic values and references that can exist are the arguments of the method under verification and the inner data of those arguments. Let $\alpha, \beta, \ldots$ denote symbolic values and references, e.g. $lit(\alpha)$ instead of $lit(z)$ and ref_α instead of ref_n.

The Local Variables

The local variables are defined as a mapping $\mathcal{M}$ where $\mathcal{M}[x \mapsto e]$ returns a new mapping with $x \mapsto e$ inserted as a new entry and $\mathcal{M}(x)$ returns the latest inserted expression e for variable x. Let writing to and reading from the stack be defined as

$$write_{stack}(\sigma, i, E) = [(pc, t, \mathcal{M}[i \mapsto E]), \ldots, \sigma_n] \textbf{ where } \sigma = [\sigma_1 = (pc, t, \mathcal{M}), \ldots, \sigma_n]$$

$$read_{stack}(\sigma, i) = \mathcal{M}(i) \textbf{ where } \sigma = [\sigma_1 = (pc, t, \mathcal{M}), \ldots, \sigma_n]$$

The Heap

The heap is defined as a mapping h from concrete reference to object structure. An object structure itself is again a mapping from each element in the object structure to the values of those elements. That is, the object structure of an instance of a class consists of its fields to their values and the object structure of an array consists of all indices to their values.

Reading from a Field A field of an object can be read from both a concrete and a symbolic reference. When reading a field from a concrete reference, the value enclosed in the field of the object structure of the corresponding concrete reference is retrieved. When reading the field of a symbolic reference, a chained if-then-else expression is returned, which accounts for every possible concrete reference the symbolic reference can be in $\mathcal{A}$.

Example 6 (Reading a field of a symbolic reference)**.** Suppose we have a SS $(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$ and the statement `value := x.f` is to be executed where the variable `x` is the symbolic reference ref_α in the SS. Suppose that the symbolic reference ref_α is mapped to the concrete references $\{ref_1, ref_2\}$ in $\mathcal{A}$.

Two reads will occur: one for the ref_1 case and one for the ref_2 case. Both cases are merged into one expression, leading to the expression

$$ite(ref_\alpha == ref_1, read_{field}(h, \emptyset, ref_1, i), read_{field}(h, \emptyset, ref_2, i))$$

□

Reading from a field is formalized as follows. Let A be the set of concrete references of ref_α retrieved from the alias map $\mathcal{A}$. Reading from a field is defined as

$$read_{field}(h, A, ref_n, i) = h(ref_n)(i)$$

$$read_{field}(h, A, ref_\alpha, i) = \begin{cases} h(ref_n)(i) & \textbf{if } A = \{ref_n\} \\ ite(ref_\alpha == ref_n, h(ref_n)(i), tail) & \textbf{if } A = \{ref_n, \dots\} \end{cases}$$

where *tail* contains the read of the other possibilities, i.e.

$$tail = read_{field}(h, A - \{ref_n\}, ref_\alpha, i)$$

Writing to a Field. A field of an object can be written to on both concrete and symbolic references. When writing to a concrete reference, the value enclosed in the field of the object structure of the corresponding concrete reference is updated. On the contrary, when writing to a symbolic reference, one must account for every possible concrete reference the symbolic reference could be and thus, a write to every possible concrete reference will occur.

Example 7 (Writing to a field of a symbolic reference)**.** Suppose we have a SS $(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$ and the statement `x.f := 1;` is to be executed where the variable `x` is the symbolic reference ref_α in the SS. Suppose that the symbolic reference ref_α is mapped to the concrete references $\{ref_1, ref_2\}$ in $\mathcal{A}$.

Two writes will be executed: one to the concrete reference ref_1 and one to the concrete reference ref_2. For both writes, the value to be assigned is changed from the right-hand side, `1`, to a conditional which constraints the write to include that the symbolic reference must be equal to

the concrete reference. The true case of this conditional contains the right-hand side and the false case contains the original value, as if no write occurs. That is, the value written to field f of ref_1 will be

$$ite(ref_\alpha == ref_1, 1, read_{field}(h, \emptyset, ref_n, i))$$

□

Writing to a field is formalized as follows. Let A be the set of concrete references of ref_α retrieved from the alias map $\mathcal{A}$. Writing to a field is defined as

$$write_{field}(h, A, ref_n, i, E) = h[ref_n \mapsto h(ref_n)[i \mapsto E]]$$

$$write_{field}(h, A, ref_\alpha, i, E) = \begin{cases} write_{field}(h, A, ref_n, i, E) & \textbf{if } A = \{ref_n\} \\ write_{field}(h', A - \{ref_n\}, E) & \textbf{if } A = \{ref_n, \dots\} \end{cases}$$

where h' contains the write to the current concrete possibility ref_n of ref_α. I.e.

$$h' = write_{field}(h, \emptyset, ref_n, i, ite(ref_\alpha == ref_n, E, read_{field}(h, \emptyset, ref_n, i)))$$

Reading an Array Element. A array element can only be read from a array pointed to by a concrete reference. However, the index may be symbolic. When reading an element from an array, the element corresponding to the index is retrieved from the object structure, similar to reading from a field. The difference between the two being that the element in the object structure may not exist, in other words, the given index is out of range. If this is the case, the function returns `infeasible`. If the given index contains a symbolic variable, all possible concrete cases must be accounted for. A chained if-then-else is constructed that contains a conditional for each possible concrete index.

Example 8 (Reading an array element with a symbolic index from a concrete array)**.** Suppose we have a SS $(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$ and the statement `x := a[i];` is to be executed, where `a` is a concrete array with object structure $\mathcal{O} = [elem_0 \mapsto 10, elem_1 \mapsto 20, elem_2 \mapsto 30]$ of size 3 and `i` is a symbolic value.

The value that is assigned to `x` will be

$$ite(i == 0, 10, ite(i == 1, 20, 30))$$

□

Reading an array element is formalized as follows. Let $s = size(h(ref_n))$ be the size of the array pointed to by reference ref_n. Note that the size of an array is always a concrete value. Let $I = \{0, \dots, s-1\}$ be the set of all indices and let $E' = eval[\![E]\!](\sigma, h)$ be the evaluated value of expression E. Then reading an array element is defined as

$$read_{elem}(\sigma, h, ref_n, E) = \begin{cases} read_{selem}(h, I, ref_n, E) & \textbf{if } E \textbf{ is a symbolic expr.} \\ h(ref_n)(elem_{E'}) & \textbf{if } 0 \leq E' < s \\ \texttt{infeasible} & \textbf{otherwise} \end{cases}$$

where reading a symbolic array element is defined as

$$read_{selem}(\sigma, h, I, ref_n, E) = \begin{cases} read_{elem}(\sigma, h, ref_n, i) & \textbf{if } I = \{i\} \\ ite(e_E == i, read_{elem}(\sigma, h, ref_n, E), tail) & \textbf{if } I = \{i, \dots, \} \end{cases}$$

where $tail = read_{selem}(\sigma, h, I - \{i\}, ref_n, E)$.

Writing to an Array Element. An array element can only be written to when performed on a concrete reference. Similar to reading an array element, the index may be symbolic. When writing to a symbolic array index, an if-then-else is written to each possible index. The if-then-else represents that the write only occurs when the symbolic reference is equal to the concrete reference.

Example 9 (Writing to an array element with a symbolic index of a concrete array)**.** Suppose we have a SS $(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$ and the statement `a[i] := x;` is to be executed, where `a` is a concrete array $\mathcal{O} = [elem_0 \mapsto 10, elem_1 \mapsto 20]$ of size 2 and `i` is a symbolic value.

Two writes will occur: one to $elem_0$ and one to $elem_1$. The values assigned to each element will be an if-then-else expression that represents that the index i must match the concrete index. We have that the expression $ite(0 == i, x, 10)$ will be assigned to $elem_0$.

□

Writing to an array element is defined as follows

$$write_{elem}(\sigma, h, ref_n, E_1, E_2) = \begin{cases} write_{selem}(\sigma, h, I, ref_n, E_1, E_2) & \textbf{if } E_1 \textbf{ is a symbolic expr.} \\ h[ref_n \mapsto h(ref_n)[elem_{E_1'} \mapsto E_2']] & \textbf{if } 0 \leq E_1' < s \\ \texttt{infeasible} & \textbf{otherwise} \end{cases}$$

where writing to a symbolic array element is defined as

$$write_{selem}(\sigma, h, I, ref_n, E_1, E_2) = \begin{cases} write_{elem}(\sigma, h, ref_n, i, v) & \textbf{if } I = \{i\} \\ write_{selem}(\sigma, h', I - \{i\}, ref_n, E_1, E_2) & \textbf{if } I = \{i, \dots\} \end{cases}$$

where $v = ite(i == E_1, E_2, read_{elem}(\sigma, h, ref_n, i))$ is the expression containing the conditional write for when index i equals the symbolic index E_1 and $h' = write_{selem}(\sigma, h, \{i\}, ref_n, E_1, E_2)$ is the updated heap containing the write to index i.

4.3.3 The Path Constraints

The path constraints, denoted by π, is the set of assumptions that must hold for the current program path to be reached. The path constraints are accumulated during the exploration. When an `assume E;` statement is executed, the boolean expression E is evaluated into one of three cases: (1) into a simplified expression E'; (2) into the constant `true`; and (3) into the constant `false`. When the evaluation results in E', the path constraints are updated to include the new assumption $\pi = \pi \cup \{E'\}$. When the evaluation results in `true`, meaning that E is valid, and thus will always hold, the exploration of this program path will continue. Finally, when the evaluation results in `false` the exploration of this program path will halt as there exists no input such that this program path will be executed.

4.3.4 The Alias Map

The alias map, denoted by $\mathcal{A}$, is a mapping from symbolic references to sets of concrete references. The alias map has several purposes. Its primary purpose is to keep track of the concrete references a symbolic reference can point to. Its secondary purpose is to determine which aliases a symbolic reference can point to when it is first initialized, as described in more detail in Section 4.4.1.

The alias map and its inhabitants are initialized lazily, inspired by [24]. The aim of lazy initialization is twofold: (1) to avoid the issues arising from cyclic data structures, e.g. a linked list; and (2) to reduce the runtime of the symbolic execution engine.

A symbolic reference is inserted in the alias map the first time it is either: (1) dereferenced via a read or write to an element or field; (2) dereferenced via the array size operator; or (3) used in a lock. When an alias map entry is initialized, the concrete reference set will be inhabited by: (1) a new concrete reference allocated on the heap with all elements of its object structure set to fresh symbolic values- and references; (2) `null`; and (3) all other concrete references pointed to by other symbolic references in $\mathcal{A}$ that are of the same type.

4.3.5 The Locks

The lock set, denoted by $\mathcal{L}$, is a mapping from references to thread ids. Each entry represents a lock being held on a reference by the thread with the corresponding thread id.

4.3.6 The Interleaving Constraints

The interleaving constraints, denoted by $\mathcal{I}$, is a set of binary relations $a \sim_G b$ and $a \not\sim_G b$, denoting that the actions a and b are independent and not independent respectively. When two actions a and b are independent, no (sub)path in which a is executed before b will be explored. The interleaving constraints are generated using partial order reduction, which is described in Section 4.4.4 in more detail.

4.4 The Symbolic Execution Algorithm

The main tasks of the algorithm are to: (1) explore the search space up to some predefined maximum program path length k, based on the structure of the CFG; (2) update the Symbolic State during the exploration; (3) add the path constraints when an assumption is encountered and try to deduce if an assumption makes a path infeasible; and (4) verify an assertion when one is encountered.

There are two modes to explore the search space. The default mode, which explores the search space by prioritizing the actions by lowest vertex label, as described in Section 4.2.1, and interleaving by lowest thread id. The random mode explores the search space randomly for the different interleavings.

The main algorithm is presented in Algorithm 1 and the execution of specific actions is presented in Subsection 4.4.1. The algorithm explores the search space until either $k = 0$ or all threads have terminated. It explores the program paths in which threads are enabled and the different interleavings which are not constrained by the interleaving constraints in $\mathcal{I}$. In the main algorithm, let *trace* denote the actions previously executed by this branch.

The algorithm is executed with the initial symbolic state:

- $\mathcal{T}$ is a singleton consisting of the main thread. In this thread, *pc* is set to the method entry of the method under verification, σ is set to a single stack frame where the program

counter to return to and the left-hand side are undefined, and the local variables are set to an unique symbolic- value or reference for each parameter;

- h is an empty mapping;
- π is the evaluated pre-condition of the method under verification;
- $\mathcal{A}$ is an empty mapping;
- $\mathcal{L}$ is an empty mapping; and
- $\mathcal{I}$ is an empty set.

Algorithm 1 The main symbolic execution algorithm

Input: The symbolic state *state* and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

```
procedure EXECUTE((T, h, π, A, L, I), k)
    if k = 0 ∨ T = ∅ then
        return valid
    else
        T_enabled ← enabled(T)
        if T_enabled = ∅ then
            return deadlock
        else
            – Construct the set of threads T_unique that need to be explored.
            T_unique ← ∅
            for all Threads T_tid = (pc, σ, η) ∈ T_enabled do
                a ← action(pc)
                if ¬∃a_i ∼_G a_j ∈ I : a_j ∈ trace ∧ a = a_i then
                    T_unique ← T_unique ∪ T_tid
                end if
            end for
            – Construct the set of new interleavings.
            A ← the actions of all a unordered pairs of T_enabled
            I_new ← por(state, A)
            I ← conflicts(I_new, I) ∪ I_new
            – Branch and continue the exploration for each thread that needs to be explored.
            for all Threads T_tid = (pc, σ, η) ∈ T_unique do
                result ← EXECUTEACTION(state, tid, k)
                if result ∈ {invalid, deadlock, unknown} then
                    return result
                end if
            end for
        end if
    end if
end procedure
```

Input: The symbolic state *state*, the current thread id *tid* and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure BRANCH($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid} \in \mathcal{T}$
  neighbours $\leftarrow N_{CFG}(pc)$
  for all Neighbours $pc' \in$ *neighbours* **do**
    $\mathcal{T}_{tid} \bullet pc \leftarrow pc'$
    result $\leftarrow$ EXECUTE($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, $k-1$)
    if *result* $\neq$ `valid` **then**
      return *result*
    end if
  end for
end procedure

4.4.1 Symbolic Execution of Actions

Let EXECUTEACTION be the procedure that selects the procedure corresponding to the given action. The actions are divided into two categories: the ones that affect the Symbolic State and those that do not. The actions that affect the SS are described in more detail below. The actions that are not described are treated as a `skip` statement, and only execute the BRANCH procedure.

Declare Statements. A declare statement defines a new variable in the current stack frame of the call stack. The value assigned to the new variable i is the default value of the corresponding type τ.

Input: The symbolic state *state*, the current thread id *tid*, the type of the variable τ, the name of the variable i and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure EXECUTEDECLARE($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, τ, i, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid} \in \mathcal{T}$
  $\mathcal{T}_{tid} \bullet \sigma \leftarrow write_{stack}(\sigma, i, default(\tau))$
  return BRANCH($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
end procedure

Assignment Statements. An assignment statement modifies the value on the left-hand side to the value on the right-hand side. The left-hand side can be either a variable, a field or an element of an array. The right-hand side can be either an expression, a field, an array element, or the instantiation of a new array. Method calls on the right-hand side are handled by first executing the method body followed by an assignment statement in which the original left-hand side is assigned to the expression right-hand side `retval`.

Assignment statements consists of three parts: (1) the (potential) initialization of symbolic reference(s); (2) the determination of the memory location of the left-hand side; and (3) the evaluation of the right-hand side.

Let us begin with the initialization of symbolic reference(s). Let *init* define the function that initializes a symbolic reference if it has not been initialized already. A symbolic reference ref_α is initialized iff the alias map $\mathcal{A}$ contains ref_α. Let *init* be defined as

$$init : Heap \times AliasMap \times SymbolicReference \rightarrow Heap \times AliasMap$$

where

$$init(h, \mathcal{A}, ref_\alpha) = \begin{cases} (h, \mathcal{A}) & \textbf{if } ref_\alpha \in \mathcal{A} \\ (h[ref_n \mapsto \mathcal{O}], \mathcal{A}[ref_\alpha \mapsto A]) & \textbf{otherwise} \end{cases}$$

where ref_n is a fresh concrete reference, $\mathcal{O}$ is the object structure of the newly allocated object, each field is set to a symbolic reference or symbolic value, and

$$A = A_{existing} \cup \{\texttt{null}, ref_{fresh}\}$$

is the set of possible concrete references ref_α can point to. Let $A_{existing}$ be the union of the sets of concrete references of all other symbolic references of the same type in $\mathcal{A}$.

Let *assign* be the function that defines the assignment of the value resulting from the right-hand side to the left-hand side and initializing a symbolic reference, when applicable. It is defined as

$$assign : Stack \times Heap \times AliasMap \times Lhs \times Expression \rightarrow Stack \times Heap \times AliasMap$$

where

$$assign(\sigma, h, \mathcal{A}, lhs_{var}(i), E_v) = (write_{stack}(\sigma, i, E_v), h, \mathcal{A})$$

$$assign(\sigma, h, \mathcal{A}, lhs_{field}(i_1, i_2), E_v) = \begin{cases} (\sigma, write_{field}(h, \emptyset, ref_n, i_2, E_v), \mathcal{A}) \\ \quad \textbf{if } read_{stack}(\sigma, i_1) = ref_n \\ \texttt{infeasible} \\ \quad \textbf{if } read_{stack}(\sigma, i_1) = \texttt{null} \\ (\sigma, write_{field}(h', \mathcal{A}'(ref_\alpha), i_2, E_v), \mathcal{A}') \\ \quad \textbf{if } read_{stack}(\sigma, i_1) = ref_\alpha, \ init(h, \mathcal{A}, ref_\alpha) = (h', \mathcal{A}') \end{cases}$$

$$assign(\sigma, h, \mathcal{A}, lhs_{elem}(i, E), E_v) = \begin{cases} (\sigma, write_{index}(h, ref_n, E, E_v), \mathcal{A}) \\ \quad \textbf{if } read_{stack}(\sigma, i) = ref_n \\ \texttt{infeasible} \\ \quad \textbf{if } read_{stack}(\sigma, i) = \texttt{null} \end{cases}$$

Let *evaluate* be the function that evaluates the right-hand side of the assignment, defined as

$$evaluate : Stack \times Heap \times AliasMap \times Rhs \rightarrow Expression \times Heap \times AliasMap$$

where

$$evaluate(\sigma, h, \mathcal{A}, rhs_{expr}(E)) = (eval[\![E]\!](\sigma, h), h, \mathcal{A})$$

$$evaluate(\sigma, h, \mathcal{A}, rhs_{field}(i, i_2)) = \begin{cases} (read_{field}(\sigma, h, \emptyset, ref_n, i_2), h, \mathcal{A}) \\ \quad \textbf{if } read_{stack}(\sigma, i_1) = ref_n \\ \texttt{infeasible} \\ \quad \textbf{if } read_{stack}(\sigma, i_1) = \texttt{null} \\ (read_{field}(\sigma, h', \mathcal{A}(ref_\alpha), ref_\alpha, i_2), h', \mathcal{A}') \\ \quad \textbf{if } read_{stack}(\sigma, i_1) = ref_\alpha, \ init(h, \mathcal{A}, ref_\alpha) = (h', \mathcal{A}') \end{cases}$$

$$evaluate(\sigma, h, \mathcal{A}, rhs_{elem}(i, E)) = \begin{cases} (read_{elem}(h, ref_n, E), h', \mathcal{A}) & \textbf{if } read_{stack}(\sigma, i) = ref_n \\ \texttt{infeasible} & \textbf{if } read_{stack}(\sigma, i) = \texttt{null} \end{cases}$$

$$evaluate(\sigma, h, \mathcal{A}, rhs_{array}(\tau, \underline{E})) = \begin{cases} (ref_n, h[ref_n \mapsto \mathcal{O}], \mathcal{A}) & \textbf{if } \underline{E} = [E_1] \\ (ref_n, h[ref_n \mapsto \mathcal{O}], \mathcal{A}) & \textbf{if } \underline{E} = [E_1, \ldots, E_n], \ \tau = array(\tau') \end{cases}$$

For the base case of ref_{array} we have that ref_n is a fresh reference, and we have the object structure of the new array $\mathcal{O} = [elem_0 \mapsto default(\tau), \ldots, elem_{E_1'} \mapsto default(\tau))$ For the recursive case, we again have that ref_n is a fresh reference, and we have the objeect structure $\mathcal{O} = [elem_0 \mapsto ref_1, \ldots, elem_{E_1'} \mapsto ref_n)$, where each $ref_1 \ldots ref_n$ are fresh references that result from recursive evaluations

$$evaluate(\sigma, h_j, \mathcal{A}, rhs_{array}(\tau', (E_1, \ldots, E_n)))$$

Finally, h_j is the heap from the previous allocated fresh reference ref_{j-1}.

Input: The symbolic state *state*, the current thread id *tid*, the left-hand side t, the right-hand side v and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteAssign($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, t, v, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid} \in \mathcal{T}$
  if $v \neq rhs_{call}(I)$ **then**
    if t or v contains a read or write from an symbolic reference ref_α which is an array **then**
      for all Concrete references $ref_n \in \mathcal{A}(ref_\alpha)$ **do**
        $\mathcal{A}' \leftarrow \mathcal{A}[ref_\alpha \mapsto \{ref_n\}]$
        $\pi' \leftarrow \pi \cup \{ref_\alpha == ref_n\}$
        result $\leftarrow$ ExecuteAssign($(\mathcal{T}, h, \pi', \mathcal{A}', \mathcal{L}, \mathcal{I})$, *tid*, t, v, k)
        if *result* $\in \{\texttt{invalid}, \texttt{deadlock}, \texttt{unknown}\}$ **then**
          return *result*
        end if
      end for
    else
      value $\leftarrow evaluate(\sigma, h, \mathcal{A}, v)$
      if *value* $= \texttt{infeasible}$ **then**
        return `infeasible`
      else if *value* $= (E_v, h', \mathcal{A}')$ **then**
        result $\leftarrow assign(\sigma, h', \mathcal{A}', t, E_v)$
        if *result* $= \texttt{infeasible}$ **then**
          return `infeasible`
        else if *result* $= (\sigma'', h'', \mathcal{A}'')$ **then**
          $\mathcal{T}_{tid} \bullet \sigma \leftarrow \sigma''$
          return Branch($(\mathcal{T}, h'', \pi, \mathcal{A}'', \mathcal{L}, \mathcal{I})$, *tid*, k)
        end if
      end if
    end if
  else
    return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
  end if
end procedure

Assert Statements. An assert statement triggers the verification of the assertion. The assertion E is combined with the path constraints π to form the formula

$$\varphi = \bigwedge_{\psi \in \pi} \psi \wedge \neg E$$

which is first evaluated. If the result is `true`, the assertion is incorrect, if it is `false`, the assertion is correct. Otherwise, if the result is a simplified formula E', the formula E' is processed further. When the formula E' contains at least one symbolic reference, a formula will be generated for each of the concrete cases. When E' contains multiple symbolic references, a formula for each combination of concrete references will be generated. Each formula will then be solved using Z3.

Input: The symbolic state *state*, the current thread id *tid*, the assertion E and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteAssert($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, E, k)
  $E' \leftarrow eval[\![\wedge_{\psi \in \pi} \psi \wedge \neg E]\!](\sigma, h)$
  if $E' = lit(\texttt{true})$ **then return** `invalid`
  else if $E' = lit(false)$ **then return** Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
  else
    result $\leftarrow$ Z3(E')
    if *result* $\neq$ `unsatisfiable` **then**
      return *result*
    else
      return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
    end if
  end if
end procedure

Assume Statements. An assume statement asserts that the assumption must be satisfiable to continue the exploration of this branch. The first step is to try solve the formula using evaluation. If the formula evaluates to `true`, the assumption always holds and the exploration of this branch continues, if it evaluates to `false`, the assumption never holds, halting further exploration of this branch. Otherwise, the simplified expression E' is added to the path constraints.

Input: The symbolic state *state*, the current thread id *tid*, the assumption E and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteAssume($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, E, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  $E' \leftarrow eval[\![E]\!](\sigma, h)$
  if $E' = lit(\texttt{false})$ **then**
    return `infeasible`
  else if $E' \neq lit(\texttt{true})$ **then**
    $\pi \leftarrow \pi \cup E'$
  end if
  return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, k)
end procedure

Return Statements. A return statement assigns the evaluated value of the expression to the special variable `retval`.

Input: The symbolic state *state*, the current thread id *tid*, the return value E and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteReturn($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, E, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  $\mathcal{T}_{tid} \bullet \sigma \leftarrow write_{stack}(\sigma, \texttt{retval}, eval[\![E]\!](\sigma, h))$
  return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, k)
end procedure

Lock Statements. A lock statement aims to acquire the lock of a reference pointed to by the variable i. This reference can be either concrete and symbolic. We need to consider three cases: (1) the reference is a symbolic reference ref_α. The SEE branches for each concrete reference of ref_α in the alias map $\mathcal{A}$. The path constraints π are updated accordingly; (2) the reference is a concrete reference ref_n. Then, if the lock reference is locked by another thread, the path is considered infeasible. If the lock is not held by another thread, the reference is added to the lock set; and (3) the reference is `null`. The program path is considered infeasible.

Input: The symbolic state *state*, the current thread id *tid*, the variable to lock i and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteLock($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, i, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  switch $read_{stack}(\sigma, i)$ **do**
    case Concrete reference ref_n**:**
      if $\mathcal{L}(ref_n) \neq tid$ **then**
        return `infeasible`
      else
        $\mathcal{L} \leftarrow \mathcal{L}[ref_n \mapsto tid]$
        return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, k)
      end if
    case Symbolic reference ref_α**:**
      for all Concrete reference $ref_n \in \mathcal{A}(ref_\alpha)$ **do**
        $\pi' \leftarrow \pi \cup \{ref_n == ref_\alpha\}$
        $\mathcal{A}' \leftarrow \mathcal{A}[ref_\alpha \mapsto \{ref_n\}]$
        $\mathcal{T}_{tid} \bullet \sigma \leftarrow \sigma[var \leftarrow ref_i]$
        result $\leftarrow$ ExecuteLock($(\mathcal{T}, h, \pi', \mathcal{A}', \mathcal{L}, \mathcal{I})$, *tid*, i, k)
        if *result* $\in \{\texttt{invalid}, \texttt{deadlock}, \texttt{unknown}\}$ **then**
          return *result*
        end if
      end for
    case `null`**:**
      return `infeasible`
end procedure

Unlock Statements. An unlock statement removes the lock of the reference ref_n pointed to by variable i from the lock set $\mathcal{L}$.

Input: The symbolic state *state*, the current thread id *tid*, the variable to unlock i and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteUnlock($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, i, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  $ref_n \leftarrow read_{stack}(\sigma, i)$
  $\mathcal{L} \leftarrow remove_{lock}(\mathcal{L}, ref_n)$
  return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, k)
end procedure

The Entry of a Method or Constructor. The entry of a method or constructor defines its entry point. The control flow is transferred from the calling method to the method body and the pre-condition of the method is verified as if it is an assert statement.

Input: The symbolic state *state*, the current thread id *tid* and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteMethodEntry($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
  if The method has a pre-condition E **then**
    Verify E in the same way as an assert statement.
  end if
  return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
end procedure

The Exit of a Method or Constructor. The exit of a method or constructor defines its exit point. The top-most stack frame is popped from the call stack and post-condition of the method is verified as if it is an assert statement.

Input: The symbolic state *state*, the current thread id *tid*, the left-hand side to assign the return value to t and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteMethodExit($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, tid, t, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  if The method has a post-condition E **then**
    Verify E in the same way as an assert statement.
  end if
  if $|\sigma| = 1$ **then**
    $\mathcal{T} \leftarrow \mathcal{T} - \{\mathcal{T}_{tid}\}$
    $h' \leftarrow h$
  else
    $retval \leftarrow read_{stack}(\sigma, \texttt{retval})$
    $\mathcal{T}_{tid} \bullet \sigma \leftarrow write_{stack}(pop_{stack}(\sigma), \texttt{retval}, retval)$
    if $\eta = [(pc', n)_1, \dots]$ **then**
      $\mathcal{T}_{tid} \bullet \eta \leftarrow [(pc', n-1)_1, \dots]$
    end if
  end if
  return Branch($(\mathcal{T}, h', \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, k)
end procedure

The Call of a Method or Constructor. The call of a method or constructor defines that a method or constructor needs to be executed. A new stack frame is pushed on the call stack, containing the parameters $\underline{P}$ of the method or constructor assigned to the arguments $\underline{E}$. When the current statements is inside a try block, the call depth of the exception handler stack η is increased by one. For constructors, the same algorithm is executed, with some modifications. A fresh reference ref_{fresh} is created. This reference is inserted on the call stack as an implicit `this` argument, similar to the other arguments supplied to the constructor. A new object structure $\mathcal{O}$ is created, which is allocated on the heap, pointed to by ref_{fresh}. The fields in the object structure $\mathcal{O}$ are set to the default values of their corresponding types.

Input: The symbolic state *state*, the current thread id *tid*, the entry point of the method pc', the parameters $\underline{P}$, the arguments $\underline{E}$, the left-hand side to assign the return value to t and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteMethodCall($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, pc', $\underline{P}$, $\underline{E}$, t, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  – *Create a new stack frame consisting of the arguments.*
  $\{pc''\} \leftarrow N_{CFG}(pc)$
  $\mathcal{T}_{tid} \bullet \sigma \leftarrow push_{stack}(\sigma, pc'', t)$
  for $j \leftarrow 1$ **to** $|P|$ **do**
    $param(\tau, i) \leftarrow \underline{P}_j$
    $\mathcal{T}_{tid} \bullet \sigma \leftarrow write_{stack}(\sigma, i, eval[\![\underline{E}_j]\!](\sigma, h))$
  end for
  – *Update the exception handler stack, if it exists.*
  if $\eta = [(pc'', n)_1, \dots]$ **then**
    $\mathcal{T}_{tid} \bullet \eta \leftarrow [(pc'', n+1)_1, \dots]$
  end if
  – *Update the program counter manually and continue the exploration.*
  $\mathcal{T}_\bullet\, pc \leftarrow pc'$
  return Execute($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, $k-1$)
end procedure

Fork Statements. The fork of a method spawn a new thread starting with at the program counter pc' with parameters $\underline{P}$ and arguments $\underline{E}$. This new thread is assigned a fresh thread id, tid_{fresh}.

Input: The symbolic state *state*, the current thread id *tid*, the entry point of the method pc', the parameters $\underline{P}$, the arguments $\underline{E}$ and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteFork($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, pc', $\underline{P}$, $\underline{E}$, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  $tid_{fresh} \leftarrow$ a fresh thread id
  $\sigma' \leftarrow push_{stack}([], \texttt{undefined}, \texttt{undefined})$
  for $j \leftarrow 1$ **to** $|P|$ **do**
    $param(\tau, i) \leftarrow \underline{P}_j$
    $\sigma' \leftarrow write_{stack}(\sigma', i, eval[\![\underline{E}_j]\!](\sigma, h))$
  end for
  $\mathcal{T}_{tid_{fresh}} \leftarrow (pc', tid, \sigma', [])$
  $\mathcal{T} \leftarrow \mathcal{T} \cup \{\mathcal{T}_{tid_{fresh}}\}$
  return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
end procedure

The Entry of a Try Block. The entry point of a try block will be executed when a try block will be entered. It inserts a new exception handler as the top-most exception handler to the η of the current thread.

Input: The symbolic state *state*, the current thread id *tid*, the catch entry block program counter pc', and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteTryEntry($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, pc', k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  $\mathcal{T}_{tid} \bullet \eta \leftarrow insert_{handler}(\eta, (pc', 0))$
  return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
end procedure

The Exit of a Try Block. The exit point of a try block will be executed when a try block will be exited. It removes the top-most exception handler from the exception handler stack η of the current thread.

Input: The symbolic state *state*, the current thread id *tid*, and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteTryExit($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  $\mathcal{T}_{tid} \bullet \eta \leftarrow remove_{handler}(\eta)$
  return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
end procedure

The Entry of a Catch Block. The entry of a catch block will be executed after an exception has occurred within a try block. The top-most exception handler will be removed from the exception handler stack η of the current thread. The control flow continues in the body of the catch block.

Input: The symbolic state *state*, the current thread id *tid*, and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteCatchEntry($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  $\mathcal{T}_{tid} \bullet \eta \leftarrow remove_{handler}(\eta)$
  return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, *tid*, k)
end procedure

The Exceptional State. The exceptional state is a special action that is reached after a `throw` has been executed. The exceptional state The exceptional state considers two cases: (1) there is a corresponding catch block the control flow will be transferred to. In this case, the stack frames of the calls made will be popped and their exceptional post-conditions will be verified; and (2) there is no corresponding catch block. In this case, the exceptional post-condition of all methods in the call stack will be verified and the exploration of this branch stops.

Input: The symbolic state *state*, the current thread id *tid*, and some integer k
Output: `valid`, `invalid`, `deadlock` or `unknown`

procedure ExecuteExceptional($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, tid, k)
  $(pc, \sigma, \eta) \leftarrow \mathcal{T}_{tid}$
  if $|\eta| = 0$ **then**
    while $|\sigma| > 0$ **do**
      if The method on top of the call stack has an exceptional post-condition E **then**
        Verify E in the same way as an assert statement.
      end if
      $\sigma \leftarrow pop_{stack}(\sigma)$
    end while
  else
    $(pc', n) \leftarrow \eta_1$
    while $n > 0$ **do**
      if The method in which pc' resides has an exceptional post-condition E **then**
        Verify E in the same way as an assert statement.
      end if
      $\mathcal{T}_{tid} \bullet \sigma \leftarrow pop_{stack}(\sigma)$
      $n \leftarrow n - 1$
    end while
    if The method has an exceptional post-condition E **then**
      Verify E in the same way as an assert statement.
    end if
    $\mathcal{T}_{tid} \bullet pc \leftarrow pc'$
    return Branch($(\mathcal{T}, h, \pi, \mathcal{A}, \mathcal{L}, \mathcal{I})$, tid, k)
  end if
end procedure

4.4.2 Formula Caching

One fairly trivial optimization is to store formulas that are sent to the SMT solver in a cache. When a formula is verified, the first step is to check if the cache contains that formula. If this is the case, the formula is valid, otherwise the SEE will have returned that the formula is invalid the first time it is sent to the SMT solver and the SEE will have terminated.

The main problem with implementing a caching approach is to control the runtime overhead. Comparing two expressions can become quite expensive, especially when expressions grow bigger.

Initial experiments showed that a set data structure based on a binary search tree [2] with $O(\log n)$ lookup time and $O(\log n)$ insert time actually reduces the overall performance of the SEE. The main reason being that it requires an ordering of expressions, which implies comparisons between different expressions.

The data structure we chose is a hash array mapped trie with $O(\log n)$ average lookup time and $O(\log n)$ average insert time. In practice, these operations are performed in $O(1)$ time. Another advantage is that the expressions are hashed, which removes the need for expensive comparisons between expressions.

4.4.3 Expression Evaluation

The Symbolic Execution Engine includes an expression evaluator. This expression evaluator aims to simplify expressions to a constant, which is possible iff the expression does not contain a symbolic reference or symbolic value. The aim of expression evaluation is threefold: (1) to determine the outcome of `assume` statements. If the outcome is a literal, the path can be pruned or the path constraint does not have to be included in the path constraints; (2) to determine the outcome of `assert` statements, which can remove the need to invoke Z3; and (3) to reduce the size of expressions in general.

The expression evaluator is defined as

$$eval[\![E]\!] : Expression \times (Stack, Heap) \to Expression$$

In the following definitions, let some primed expression E' be the evaluated expression of E. That is, $E' = eval[\![E]\!](\sigma, h)$.

Literals and References. Let the expression evaluator be the identity function for the literals and the references.

$$\begin{aligned} eval[\![lit(z)]\!](\sigma, h) &= lit(z) \\ eval[\![lit(b)]\!](\sigma, h) &= lit(b) \\ eval[\![lit(\alpha)]\!](\sigma, h) &= lit(\alpha) \\ eval[\![ref_n]\!](\sigma, h) &= ref_n \\ eval[\![ref_\alpha]\!](\sigma, h) &= ref_\alpha \\ eval[\![\texttt{null}]\!](\sigma, h) &= \texttt{null} \end{aligned}$$

Variable Access. Let the expression evaluator read the variable i from the stack σ when reading a variable.

$$eval[\![var(i)]\!](\sigma, h) = read_{stack}(\sigma, i)$$

Unary Operators. Let the expression evaluator apply the unary operator when the operand is a literal.

$$eval[\![unop(E, \texttt{!})]\!](\sigma, h) = \begin{cases} lit(\neg b) & \textbf{if } E' = lit(b) \\ unop(E', \texttt{!}) & \textbf{otherwise} \end{cases}$$

$$eval[\![unop(E, \texttt{-})]\!](\sigma, h) = \begin{cases} lit(-z) & \textbf{if } E' = liz(z) \\ unop(E', \texttt{-}) & \textbf{otherwise} \end{cases}$$

Arithmetic Operators. Let the expression evaluator apply the binary arithmatic operator when both operands are literals.

$$eval[\![binop(E_1, \odot, E_2)]\!](\sigma, h) = \begin{cases} lit(z_1 \odot z_2) & \textbf{if } E_1' = lit(z_1) \textbf{ and } E_2' = lit(z_2) \\ binop(E_1', \odot, E_2') & \textbf{otherwise} \end{cases}$$

Logical Operators. Let all binary logical operators be defined in terms of logical conjunction. Let the expression evaluator apply the logical conjunction operator be when both operands are literals. Otherwise, if either operand results in `true` the expression evaluator results in the simplified expression of the other operand.

$$eval[\![binop(E_1, \texttt{\&\&}, E_2)]\!](\sigma, h) = \begin{cases} lit(b_1 \wedge b_2) & \textbf{if } E_1' = lit(b_1) \textbf{ and } E_2' = lit(b_2) \\ E_2' & \textbf{if } E_1' = lit(\texttt{true}) \\ E_1' & \textbf{if } E_2' = lit(\texttt{true}) \\ lit(\texttt{false}) & \textbf{if } E_1' = lit(\texttt{false}) \textbf{ or } E_2' = lit(\texttt{false}) \\ binop(E_1', \texttt{\&\&}, E_2') & \textbf{otherwise} \end{cases}$$

$$eval[\![binop(E_1, \texttt{||}, E_2]\!](\sigma, h) = eval[\![unop(binop(unop(E_1, \texttt{!}), \texttt{\&\&}, unop(E_2, \texttt{!})), \texttt{!})]\!](\sigma, h)$$

$$eval[\![binop(E_1, \texttt{==>}, E_2]\!](\sigma, h) = eval[\![binop(unop(E_1, \texttt{!}), \texttt{||}, E_2)]\!](\sigma, h)$$

Equality and Inequality Operators. Let inequality be defined in terms of equality. Let the expression evaluator apply the the equality operator when both operands are either literals or references.

$$eval[\![binop(E_1, \texttt{==}, E_2)]\!](\sigma, h) = \begin{cases} lit(z_1 = z_2) & \textbf{if } E_1' = lit(z_1) \textbf{ and } E_2' = lit(z_2) \\ lit(b_1 = b_2) & \textbf{if } E_1' = lit(b_1) \textbf{ and } E_2' = lit(b_2) \\ lit(ref_a = ref_b) & \textbf{if } E_1' = ref_a \textbf{ and } E_2' = ref_b \\ binop(E_1', \texttt{==}, E_2') & \textbf{otherwise} \end{cases}$$

$$eval[\![binop(E_1, \texttt{!=}, E_2)]\!](\sigma, h) = eval[\![unop(binop(E_1, \texttt{==}, E_2), \texttt{!})]\!](\sigma, h)$$

Comparison Operators. Let all comparison operators be defined in terms of `<`. Let the expression evaluator apply the comparison operator when both operands are literals.

$$eval[\![binop(E_1, \texttt{<}, E_2)]\!](\sigma, h) = \begin{cases} lit(z_1 < z_2) & \textbf{if } E_1' = lit(z_1) \textbf{ and } E_2' = lit(z_2) \\ binop(E_1', \texttt{<}, E_2') & \textbf{otherwise} \end{cases}$$

$$eval[\![binop(E_1, \texttt{<=}, E_2)]\!](\sigma, h) = eval[\![unop(binop(E_2, \texttt{<}, E_1), \texttt{!})]\!](\sigma, h)$$

$$eval[\![binop(E_1, \texttt{>}, E_2)]\!](\sigma, h) = eval[\![binop(E_2, \texttt{<}, E_1)]\!](\sigma, h)$$

$$eval[\![binop(E_1, \texttt{>=}, E_2)]\!](\sigma, h) = eval[\![unop(binop(E_1, \texttt{<}, E_2), \texttt{!})]\!](\sigma, h)$$

Sizeof Operator. Let the expression evaluator return the size of the array pointed to by the variable i.

$$eval[\![sizeof(i)]\!](\sigma, h) = lit(size(h(read_{stack}(\sigma, i))))$$

If-Then-Else Operator. Let the expression evaluator return the true expression E_2' when the guard E_1' evaluates to `true` and let the false expression E_3' be returned when the guard evaluates to `false`.

$$eval[\![ite(E_1, E_2, E_3)]\!](\sigma, h) = \begin{cases} E_2' & \textbf{if } E_1' = lit(\texttt{true}) \\ E_3' & \textbf{if } E_1' = lit(\texttt{false}) \\ ite(E_1', E_2', E_3') & \textbf{otherwise} \end{cases}$$

Quantifiers. Let the expression evaluator return an evaluated chain of the respective logical operators.

$$eval[\![forall(i_1, i_2, i_3, E)]\!](\sigma, h) = eval[\![\&\&_{j=0}^{n-1} eval[\![E]\!](\sigma_j, h)]\!](\sigma, h)$$
$$eval[\![exists(i_1, i_2, i_3, E)]\!](\sigma, h) = eval[\![||_{j=0}^{n-1} eval[\![E]\!](\sigma_j, h)]\!](\sigma, h)$$

where j represents the indices $[0, \ldots, n)$ of the array retrieved from the object structure

$$read_{stack}(\sigma, i_3)) = \mathcal{O} = [elem_0 \mapsto E_0, \ldots, elem_n \mapsto E_n)$$

and

$$\sigma_j = write_{stack}(write_{stack}(\sigma, i_1, E_j), i_2, j)$$

represents the stack with the current index j and element at index j written to the stack.

4.4.4 Partial Order Reduction

Partial Order Reduction (POR) is a technique to reduce the number of different interleavings and thus aims to limit the effects of the path explosion problem as a result of concurrency. POR exploits the fact that the interleaving of actions of different threads is a partial order, in which some orders result in the same state as others.

Suppose we have a simple program, as shown in Figure 4.5, consisting of two methods `foo` and `bar`, which are executed concurrently. Suppose `Num` is a class that contains the field `value`. The complete interleaving forms the partial order as shown in Figure 4.6. The partial order consists of three paths: p_1, p_2 and p_3, each corresponding to an unique interleaving.

$$p_1 = 1; 2; a; \qquad p_2 = 1; a; 2; \qquad p_3 = a; 1; 2;$$

Paths p_2 and p_3 always lead to the same state, while p_1 and p_2 lead to the same state iff `num` points to a different reference. POR aims to identify these equivalences and prune those paths that lead to the same state.

Figure 4.5: A program consisting of two threads.

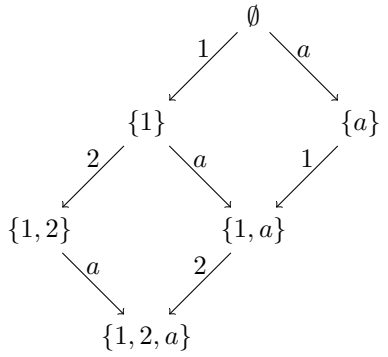
Listing 4.4: Thread 1

```
    void foo(Num num) {
1.      int y := 1;
2.      num.value := y;
    }
```

Listing 4.5: Thread 2

```
    void bar(Num num) {
a.      num.value := 2;
    }
```

Figure 4.6: The partial order of interleavings of the two threads in Figure 4.5. Each path from top to bottom represents an unique interleaving.



Conventional Partial Order Reduction uses the notion of an independence relation [18].

Definition 9 (Independence Relation)**.** $\sim \subseteq \mathit{Action} \times \mathit{Action}$ is an independence relation iff for each $(a_1, a_2) \in \sim$ the following two properties hold for all states $\mathit{state} \in \mathit{SymbolicState}$:

1. if a_1 is enabled in state and $\mathit{state} \xrightarrow{a_1} \mathit{state}'$, then a_2 is enabled in state iff a_2 is enabled in state'; and
2. if a_1, a_2 are enabled in state, there is a unique state state' such that $\mathit{state} \xrightarrow{a_1;a_2} \mathit{state}'$ and $\mathit{state} \xrightarrow{a_2;a_1} \mathit{state}'$.

These two properties formalize that (1) two independent actions cannot enable or disable each other and that (2) the order in which these two actions are executed is irrelevant, i.e. they are commutative.

Our approach to Partial Order Reduction, inspired by [34], uses the notion of guarded independence.

Definition 10 (Guarded Independence Relation)**.** $\sim_G \subseteq \mathit{Action} \times \mathit{Action}$ is a guarded independence relation with respect to a condition c_G iff c_G implies that the following two properties hold:

1. if a_1 is enabled in state and $\mathit{state} \xrightarrow{a_1} \mathit{state}'$, then a_2 is enabled in state iff a_2 is enabled in state'; and
2. if a_1, a_2 are enabled in state, there is a unique state state' such that $\mathit{state} \xrightarrow{a_1;a_2} \mathit{state}'$ and $\mathit{state} \xrightarrow{a_2;a_1} \mathit{state}'$.

stating that, instead of defining the independence relation in terms of all possible states, the independence relation is defined in terms of a formula c_G.

The formula c_G of two actions a_1 and a_2 is defined using the following two functions

$$R[\![a]\!] : \mathit{Action} \times \mathit{SymbolicState} \to \mathcal{P}(\mathit{Reference})$$

and

$$W[\![a]\!] : \mathit{Action} \times \mathit{SymbolicState} \to \mathcal{P}(\mathit{Reference})$$

which define the set of references read from and written to by the action respectively. The formula c_G is defined as the conjunction of the following three equalities

$$W[\![a_1]\!](state) \cap W[\![a_2]\!](state) = \emptyset$$
$$R[\![a_1]\!](state) \cap W[\![a_2]\!](state) = \emptyset$$
$$R[\![a_2]\!](state) \cap W[\![a_1]\!](state) = \emptyset$$

in other words, do actions a_1 and a_2 both write, or read and write, to the same region of the heap in the Symbolic State.

Implementation in Symbolic Execution

Partial Order Reduction (POR) is typically used in the context of symbolic- and explicit model checking [34, 35, 18], but it also appears in Symbolic Execution (SE) [20]; which uses an approach where they keep track of the independence using a backtracking set.

The guarded independence relation is used to determine which interleaving to explore and which to drop. Let the independence relation $a1 \sim_G a_2$ denote that any (sub)path containing the sequence of actions $a_2; \ldots; a_1;$ will not be explored.

Let *por* be the function that constructs the interleaving constraints from the symbolic state *state* and all unordered pairs of actions of all threads A.

$$por(state, A) = \bigcup_{\{a_i, a_j\} \in A} independent(state, a_i, a_j)$$

Then let *independent* determine whether two actions are independent, using the definition of c_G as described above.

$$independent(state, a_i, a_j) = \begin{cases} \{a_i \sim_G a_j\} & \textbf{if } c_G \textbf{ holds for } a_i \textbf{ and } a_j \\ \{a_i \not\sim_G a_j\} & \textbf{otherwise} \end{cases}$$

Two interleaving constraints $a_i \sim_G a_j$ and $a_p \not\sim_G a_q$ are in conflict when either $a_i = a_q$ or $a_j = a_p$. The *conflicts* function removes such conflicts from an interleaving constraint set $\mathcal{I}$ given some new interleaving constraints $\mathcal{I}_{new}$.

$$conflicts(\mathcal{I}_{new}, \mathcal{I}) = \begin{cases} \{a_i \sim_G a_j\} \cup conflicts(\mathcal{I}_{new}, \mathcal{I} - \{a_i \sim_G a_j\}) & \textbf{if } a_i \sim_G a_j \in \mathcal{I} \\ \{a_i \not\sim_G a_j\} \cup conflicts(\mathcal{I}_{new}, \mathcal{I} - \{a_i \not\sim_G a_j\}) & \textbf{if } a_i \not\sim_G a_j \in \mathcal{I} \\ & \textbf{, } \neg conflict(\mathcal{I}_{new}, a_i \not\sim_G a_j) \\ conflicts(\mathcal{I}_{new}, \mathcal{I} - \{a_i \not\sim_G a_j\}) & \textbf{if } a_i \not\sim_G a_j \in \mathcal{I} \\ & \textbf{, } conflict(\mathcal{I}_{new}, a_i \not\sim_G a_j) \\ \emptyset & \textbf{otherwise} \end{cases}$$

where the conflict between two independence relations is given by

$$conflict(\mathcal{I}_{new}, a_i \not\sim_G a_j) = \exists a_p \sim_G a_q \in \mathcal{I}_{new} : a_i = a_q \vee a_j = a_p$$

Example 10 (POR applied to the example in Figure 4.5.)**.** Suppose we have the program as shown in Figure 4.5, where ref_1 is supplied as an argument to `num` in both threads. The Symbolic Execution can be seen in Figure 4.7, where states with a dashed border will be pruned due to

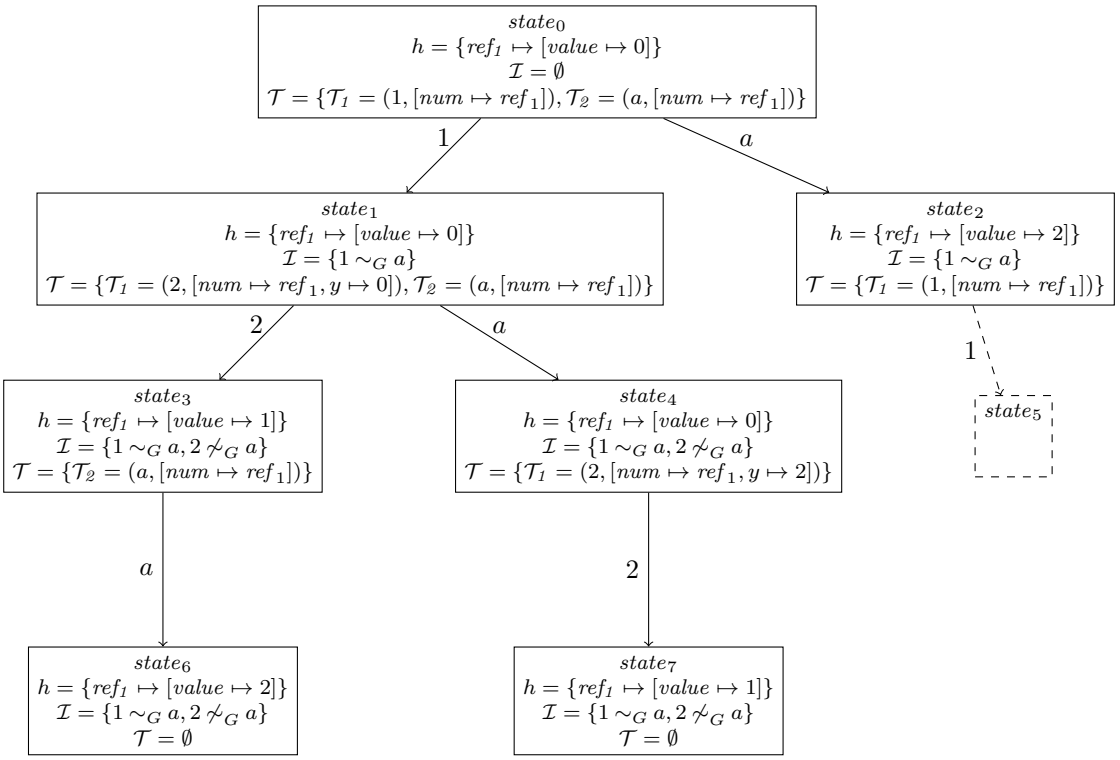
POR. Simplified versions of the states will be presented; thread only consists of a program counter and its local variables and the path constraints, alias map and lock set are omitted.

Suppose that the symbolic execution starts with the symbolic state $state_0$. Both threads need to be explored, as there are no interleaving constraints at this point. The first step is to compute the new interleaving constraints of all unordered pairs of actions. That is, actions 1 and action a. They do not write to the same part to the heap, i.e. c_G holds for a and 1. Thus, the interleaving constraint $1 \sim_G a$ is generated and is added to $\mathcal{I}$.

When branching from $state_1$, the first step is to check if there exists an interleaving constraint $1 \sim_G a \in \mathcal{I}$ that implies that action 2 does not have to be explored. This interleaving constraint does not exist, so the interleaving will be explored. The formula c_G does not hold for 2 and a, as they both write to ref_1, meaning that the independence relation $2 \not\sim_G a$ will be added to the interleaving constraints.

When branching from $state_2$, the first step is to check if there exists and interleaving constraint $1 \sim_G a \in \mathcal{I}$. This relation exists, implying that this branch does not have to be explored, as it will lead to the same state as another interleaving.

Figure 4.7: The search space of the program in Figure 4.5, including the Symbolic State. A state with a dashed border denotes that the branch will not be explored.



Chapter 5

Results

To evaluate our approach, we developed a prototype of the Symbolic Execution Engine based on the description in Chapter 4. This prototype supports all of the features as described, except for:

- Non-static methods, these can be modelled using static methods;[1]
- Any primitive typed variable other than booleans and integers; and
- Assigning a new array of symbolic size to a variable.

The goal of the experiments are to verify the completeness, the soundness and to measure the performance. These goals has been split into seven categories, as described in Table 5.1.

We performed all experiments on a Hyper-V virtual machine with a 64-bit Ubuntu 19.10 operating system. The machine used an Intel Core i5-4670 processor and was allocated with 7 gigabyte of memory.

Table 5.1: An overview of the goals of individual experiments.

	EXP-1	EXP-2	EXP-3	EXP-4	EXP-5	EXP-6	EXP-7	EXP-8
Soundness							X	X
Completeness	X	X	X	X				
Soundness of Optimizations	X	X	X	X				
Completeness of Optimizations	X	X	X	X				
Efficacy of Optimizations	X	X	X	X				
Scalability					X	X		
Relative Performance							X	X

To verify the soundness and completeness of the approach and optimizations, a mutation operator

[1]Static methods can model non-static methods due to the fact that OOX does not have dynamic dispatch. Suppose that inheritence will be supported. Then dynamic dispatch will be required and static methods can no longer model non-static methods.

scheme is designed, which is presented in Table 5.3. A mutant is generated for each point in the program where a mutation operator can be applied.

To compare the effect of individual optimizations, all possible combinations of optimizations will be compared. These combinations are presented in Table 5.2.

Table 5.2: The following scenarios will be compared. Note that when the test case is a non-concurrent program, partial order reduction is irrelevant and different cases collapse. That is A=/rand=A/def=SC, PS=S, PC=C and P=N.

	A/rand	A/def	PS	PC	SC	P	S	C	N
Partial Order Reduction (P)	X	X	X	X		X			
Expression Evaluation (S)	X	X	X		X		X	X	
Formula Caching (C)	X	X		X	X			X	
Interleaving Exploration (ran/def)	R	D	D	D	D	D	D	D	D

Table 5.3: Mutation operator scheme. When a point in the program matches the pattern in the second column of the table, a mutant is generated for each option in the third column. Let ϵ denote the removal of that specific point in the program.

Mutation operators of statements			
DEL	`S;`	ϵ	Statement removal.
FLOW	`continue;` `break;`	`break;` `continue;`	Mutation of control flow breaking statements.
Mutation of concurrency statements [7]			
FORK	`fork f(a, b);`	`f(a, b)`	Mutation of a fork into a regular call.
LOCK	`lock(x) { S; }`	`lock(y) { S; } { S; }`	Mutation of a lock to lock a different (existing) reference or the removal of the lock.
Mutation operators of variables			

VAR	`x`
	`x.f`
	`x[e]`
		`y`
		`y.f`
		`y[e]`
			Mutation of a variable to a different (existing) variable of the same type.

Mutation operators of literals			

LIT	`n`
	`true`
	`false`
		`n+1 n-1`
		`false`
		`true`
			Mutation of an integer or boolean literal.

Mutation operators of binary operators			
EQ	`==` `!=`	`!=` `==`	Mutation of the equality operators.

CMP	`<`
	`<=`
	`>`
	`>=`
		`<= > >=`
		`< > >=`
		`< <= >=`
		`< <= >`
			Mutation of the comparison operators.

ARITH	`+`
	`-`
	`*`
	`/`
	`%`
		`- * / %`
		`+ * / %`
		`+ - / %`
		`+ - * %`
		`+ - * /`
			Mutation of the arithmetic operators.

BOOL	`||`
	`&&`
	`==>`
		`&& ==>`
		`|| ==>`
		`|| &&`
			Mutation of the boolean operators.

Mutation operators of the unary operators			
UN	`!` `-`	ϵ ϵ	Unary operator removal.

5.1 The Completeness, Soundness and Efficacy of the Optimizations

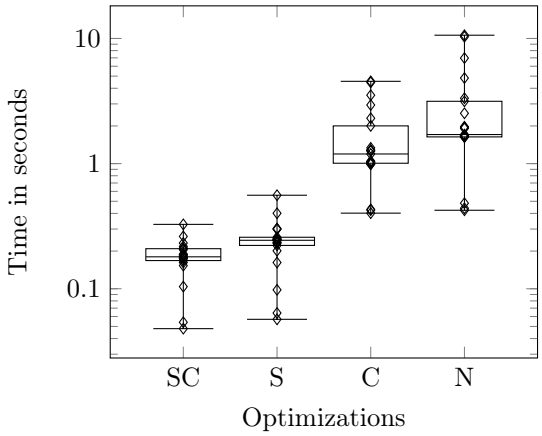
The first part of the experiments aims to evaluate the completeness of the Symbolic Execution Engine, and the completeness, soundness and efficacy of the optimizations.

EXP-1: Bubblesort. The first experiment consists of the verification of a bubblesort algorithm. The algorithm is described in Listing B.1. The algorithm is checked to be valid with respect to its specification, that is: given a non-null array of integers, the return value of the method is an array of integers in ascending order and the method will not terminate with an exception. The maximum program depth we used is 80 and the maximum size of the symbolic arrays is limited to 3.

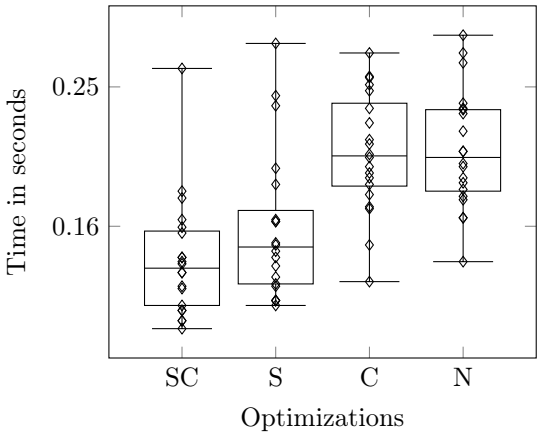
The mutation test generator created 61 mutants of which 20 resulted in a valid specification of which 8 were non-terminating and 12 were due to a specification that is not strong enough, these were verified by hand; 21 resulted in an invalid ensures specification; and 20 resulted in an invalid exceptional specification.

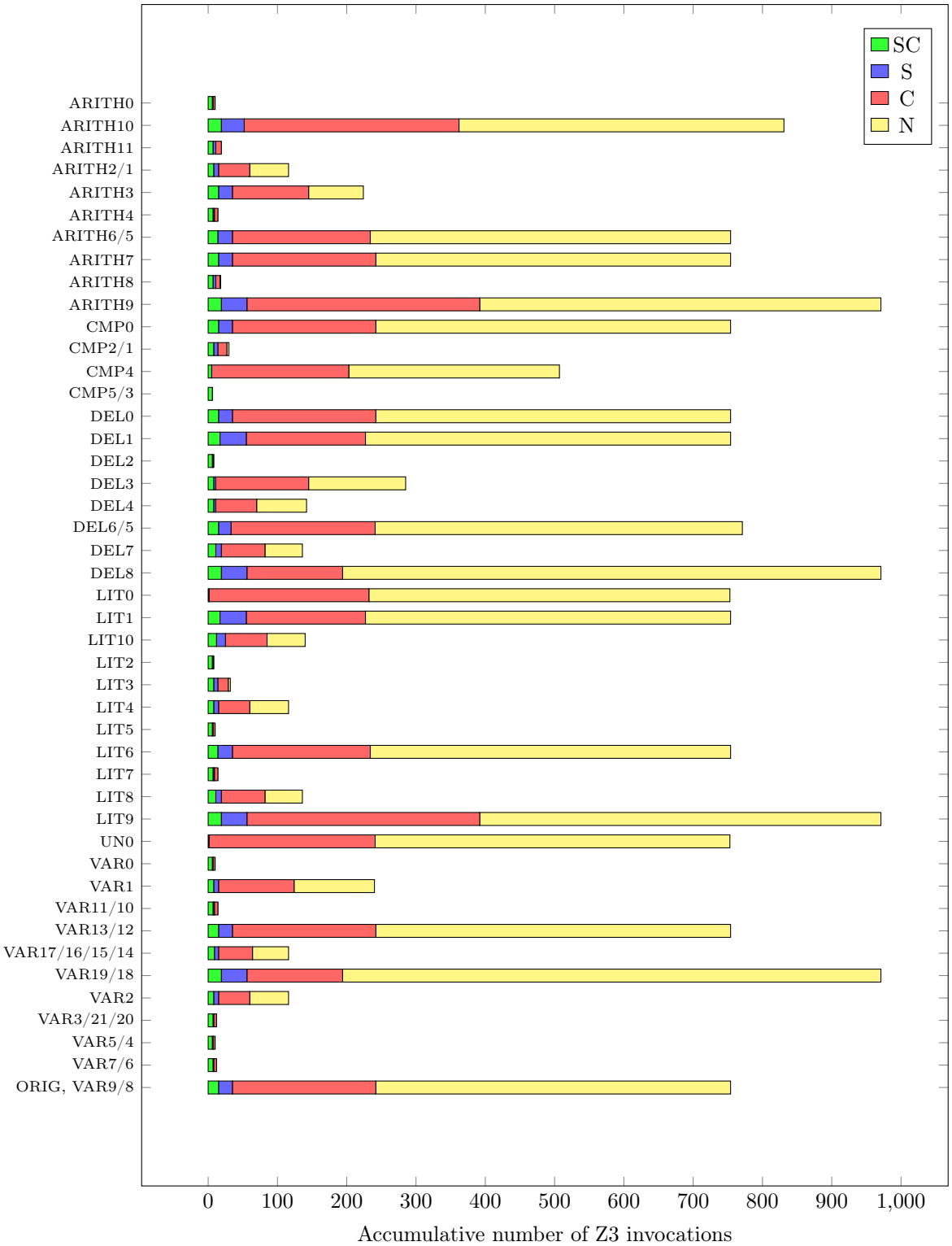
The results clearly show that the optimizations can drastically reduce the runtime. Formula caching decreases the runtime by a factor of 5 and in some cases about 25% of the formulas are duplicate, and can be retrieved from the cache. The expression evaluation also greatly contributes, in some cases decreasing the runtime by a factor of 7.

Runtime of the mutants that resulted in an invalid ensures specification.



Runtime of the mutants that resulted in an invalid exceptional specification.

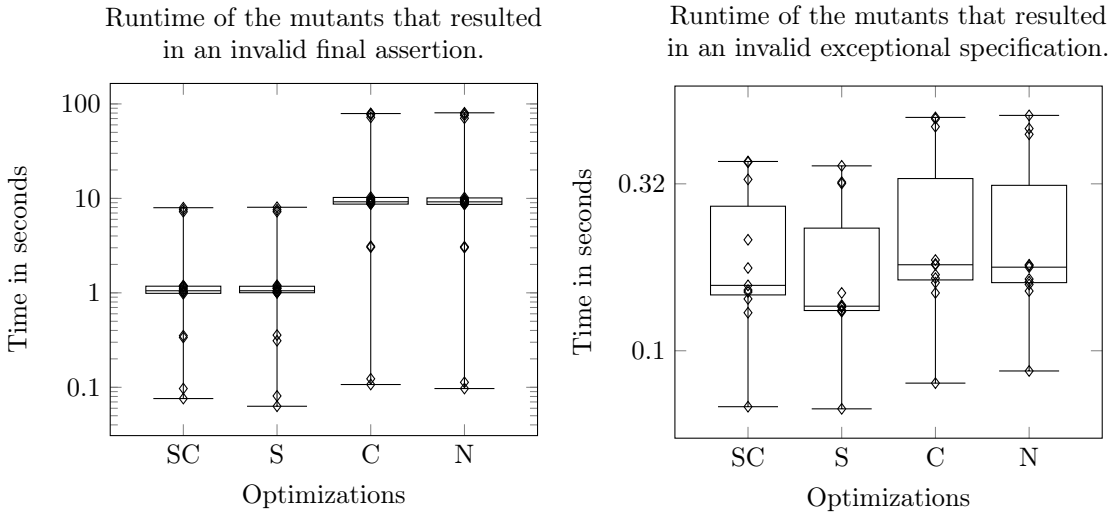



SC
S
C
N
ARITH0
ARITH10
ARITH11
ARITH2/1
ARITH3
ARITH4
ARITH6/5
ARITH7
ARITH8
ARITH9
CMP0
CMP2/1
CMP4
CMP5/3
DEL0
DEL1
DEL2
DEL3
DEL4
DEL6/5
DEL7
DEL8
LIT0
LIT1
LIT10
LIT2
LIT3
LIT4
LIT5
LIT6
LIT7
LIT8
LIT9
UN0
VAR0
VAR1
VAR11/10
VAR13/12
VAR17/16/15/14
VAR19/18
VAR2
VAR3/21/20
VAR5/4
VAR7/6
ORIG, VAR9/8
0
100
200
300
400
500
600
700
800
900
1,000
Accumulative number of Z3 invocations

EXP-2: Finding the Minimum Element in a Linked List. The second experiment aims to verify an algorithm that finds the minimum element stored in a linked list data structure. The algorithm, as described in Listing B.4, contains some auxiliary verification code: the variable `N` to determine the maximum depth of the linked list, the array `values` to keep track of all values in the linked list, and the final assertion to verify if the minimum element actually is the minimum element. The maximum program depth we used is 85.

The mutation test generator created 47 mutants of which 18 were valid of which 8 were non-terminating and 10 were due to an specification that is not strong enough, these were verified by hand; 18 resulted in an invalid final assertion; and 11 resulted in an invalid exceptional specification.

The results show that the expression evaluation optimization decreases the runtime by up to a factor of 10, from 10 seconds to 100 seconds. The formula caching increases the runtime by a tiny amount, but its effect is negligible.

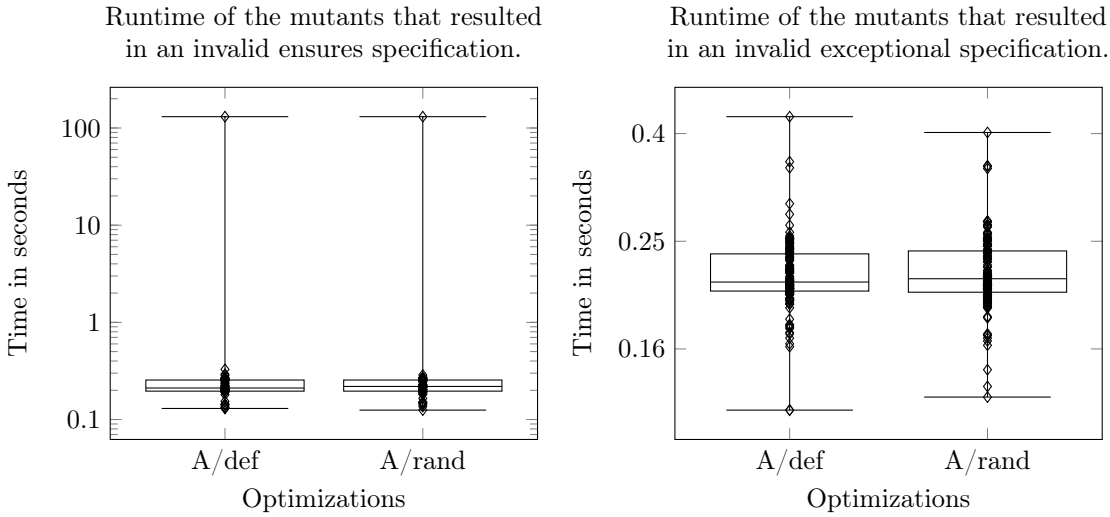


EXP-3: Concurrent Mergesort. The third experiment aims to verify a concurrent mergesort algorithm. It is described in Listing B.5. It is a concurrent version of the mergesort algorithm with a fork-join approach. The algorithm is valid with respect to its specification, that is: given a non-null array of integers, the return value of the method is an array of integers in ascending order and the method will not terminate with an exception. The maximum program depth we used is 125 and the maximum size of the symbolic arrays is limited to 3.

The mutation test generator created 480 mutants of which 278 resulted in a valid specification; 139 resulted in an invalid ensures specification; and 63 resulted in an invalid exceptional specification.

The experiments show that there is a small performance gain in using randomized path selection over default path selection, but this effect is minimal. There is one outlier (CMP2), which has a runtime 130 seconds. This mutant has an extreme runtime compared to the others while terminating with a correct result. This mutant has an extreme runtime because of the order in which the symbolic array are concretized. A concrete array of size 1 gets explored before size 2. An array of size 1 causes infinite recursion in which, at every recursive step, a new thread gets

spawned. This causes many different interleavings that have to be explored. For an array of size 2, the algorithm terminates immediately with the correct result.



EXP-4: The Dining Philosophers Problem. The fourth experiment aims to find the deadlock in the algorithm for the dining philosophers problem. The algorithm to (incorrectly) handle the problem is given in Listing B.7. It is expected that the algorithm results in a deadlock. The program contains an auxiliary variable `n` which denotes the number of philosophers, and thus the number of child threads of the main thread. The eating of philosophers is modelled using locks. The maximum program depth we used is 100.

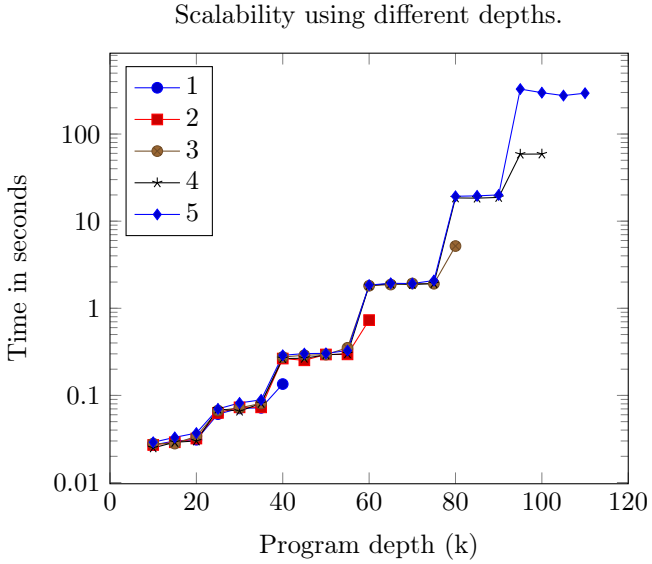
The program was executed 50 times using two modes: A/rand and A/def. Both modes correctly found that the program result in a deadlock. The default mode finds the deadlock in an average time of 0.42 seconds with a standard deviation of 0.045. The random mode finds the deadlock in 0.701 seconds with a standard deviation of 0.363 seconds.

5.2 Scalability

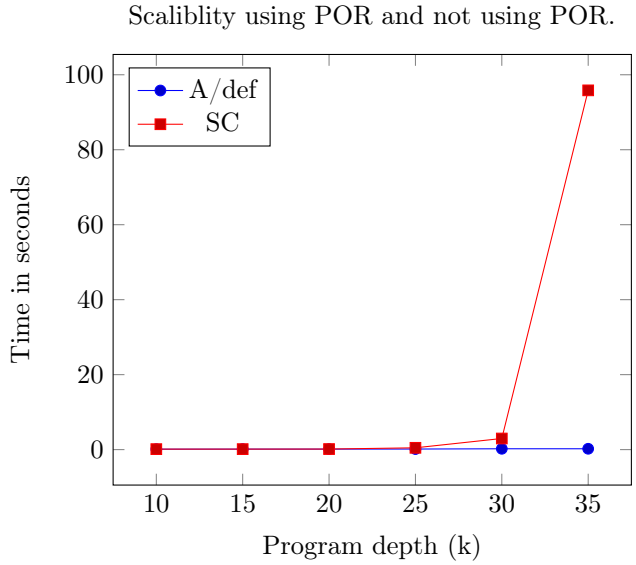
The second part of the experiments aims to determine the scalability of our approach.

EXP-5: Finding the Minimum Element in a Linked List. The fifth experiment again uses the algorithm to find the minimum element in a linked list. In this instance to determine the limits of the maximum program depth k. A maximum runtime of 300 seconds was given and the runtime for the different linked lists of length 1 up to 5 are compared.

The experiment shows that the length of the linked list does not affect the performance; the maximum depth does, as expected. The limit of the maximum program depth was 110, after which the runtime was more than 300 seconds. Most of the runtime is due to the number invocations of Z3. The number of invocations required increases exponentially relative to the maximum program depth.



EXP-6: Concurrent Mergesort. The sixth experiment again uses the concurrent mergesort algorithm. The algorithm is used to check the need and effectivity of Partial Order Reduction. The results show that when multiple threads exist, POR is a necessity. The runtime is linear up to program depth 35. After program depth 35, multiple threads begin to exist, drastically increasing the runtime. When using POR, the runtime does not show this drastic increase.



5.3 A Comparison with CBMC and CIVL

We compared our approach with CBMC [11] and CIVL [32]. CBMC is a symbolic model checker for (non-concurrent) C programs and CIVL is a symbolic execution engine for (concurrent) C programs. See Section 6.2 for more information about CIVL and CBMC.

A comparison between OOX and C cannot be completely fair. For example, when verifying C programs, one has to account for low-level pointer arithmetic. To remain sound, the Symbolic Execution Engine also needs to account that a pointer can point to every possible position in the memory. The type system of C can not be of much help either, as the C type system is fairly weak. Our aim is to make the comparison as fair as possible. We tweaked both CBMC and CIVL to halt on the first specification violation, and only verify user-defined assertions, division by zero, and array bounds. The commands we used are

```
cbmc -bounds-check -div-by-zero-check -stop-on-fail -depth <depth> <file>

civl verify -checkMemoryLeak=false -maxDepth=<depth> <file>
```

where `<depth>` is the maximum program path length, which we set equal to their OOX counterparts and `<file>` is the file under verification. The default SMT solver of CMBC is MiniSAT [15]. Early experiments showed that CBMC performed better using MiniSAT than using Z3. We used Z3 [13] as the SMT solver for CIVL. For each experiment, the original OOX program and the invalid mutations (or a subset of mutations) were translated to a version for CBMC and a version for CIVL. The translated programs were implemented to be as similar to their OOX counterparts as possible. We executed each experiment 5 times, similar to the OOX experiments.

EXP-7: Bubblesort. The seventh experiment consists of comparing our results of the bubblesort algorithm with results of CBMC and CIVL. The algorithm is translated to two other versions: a version in C for CBMC, as presented in Listing B.2 and a version in CIVL-C, as presented in Listing B.3.

The results show that our approach is sound; the results are the same as that of CBMC and CIVL. Our approach shows to have a competitive runtime in comparison.

Table 5.4: The comparative results of the bubblesort algorithm between OOX, CBMC and CIVL.

File	OOX		CBMC		CIVL	
	Expec. Result	Time (s)	Equiv. Result	Time (s)	Equiv. Result	Time (s)
ORIGINAL	VALID	0.342	✓	1.460	✓	2.96
ARITH0	INVALID	0.130	✓	0.541	✓	2.15
ARITH1	INVALID	0.205	✓	0.560	✓	2.16
ARITH2	INVALID	0.178	✓	0.559	✓	2.17
ARITH3	INVALID	0.327	✓	0.666	✓	2.49
ARITH4	INVALID	0.178	✓	0.578	✓	2.18
ARITH8	INVALID	0.155	✓	0.677	✓	2.28
ARITH11	INVALID	0.143	✓	0.633	✓	2.34
CMP1	INVALID	0.180	✓	0.575	✓	2.46
CMP2	INVALID	0.176	✓	0.567	✓	2.80
CMP3	INVALID	0.174	✓	0.578	✓	2.09
CMP4	INVALID	0.104	✓	0.589	✓	2.24
CMP5	INVALID	0.120	✓	0.601	✓	2.09
DEL2	INVALID	0.113	✓	43.098	✓	2.06
DEL3	INVALID	0.169	✓	0.644	✓	2.21
DEL4	INVALID	0.153	✓	0.666	✓	2.23
DEL7	INVALID	0.218	✓	0.272	✓	2.42
LIT0	INVALID	0.048	✓	0.204	✓	2.19
LIT2	INVALID	0.122	✓	0.633	✓	2.10
LIT3	INVALID	0.162	✓	0.570	✓	2.30
LIT4	INVALID	0.168	✓	0.528	✓	2.36
LIT5	INVALID	0.116	✓	0.636	✓	2.20
LIT7	INVALID	0.143	✓	0.660	✓	2.26
LIT8	INVALID	0.232	✓	0.303	✓	2.38
LIT10	INVALID	0.262	✓	0.543	✓	2.35
UN0	INVALID	0.054	✓	0.205	✓	2.07
VAR0	INVALID	0.116	✓	32.603	✓	2.14
VAR1	INVALID	0.175	✓	9.543	✓	2.14
VAR2	INVALID	0.184	✓	0.602	✓	2.12
VAR3	INVALID	0.140	✓	33.697	✓	2.10
VAR4	INVALID	0.120	✓	6.517	✓	2.29
VAR5	INVALID	0.267	✓	6.367	✓	2.35
VAR6	INVALID	0.136	✓	6.514	✓	2.32
VAR7	INVALID	0.141	✓	6.733	✓	2.19
VAR10	INVALID	0.136	✓	6.609	✓	2.19
VAR11	INVALID	0.129	✓	6.787	✓	2.31
VAR14	INVALID	0.189	✓	0.185	✓	2.34
VAR15	INVALID	0.211	✓	0.601	✓	3.45
VAR16	INVALID	0.209	✓	0.200	✓	2.18
VAR17	INVALID	0.188	✓	0.573	✓	3.30
VAR20	INVALID	0.162	✓	9.156	✓	2.28
VAR21	INVALID	0.158	✓	8.991	✓	2.27

EXP-8: Concurrent Mergesort. The eight experiments consists of comparing our results of the concurrent mergesort algorithm with results of CBMC and CIVL. The concurrent mergesort algorithm is translated to one other version: a version in CIVL-C, as presented in Listing B.6.

The results again show that our approach is sound; the results are the same as that of CIVL. There is one anomaly: the extreme runtime difference of CMP2. We contacted the researchers of CIVL, but they could not help in explaining this difference. They mentioned that "CIVL might be lucky".

Table 5.5: The comparative results of the concurrent mergesort algorithm between OOX and CIVL.

File	**OOX**		**CIVL**	
	Expec. Result	**Time (s)**	**Equiv. Result**	**Time (s)**
ORIGINAL	VALID	1.194	✓	4.43
CMP2	INVALID	130.765	✓	4.06
LIT14	INVALID	0.189	✓	3.55
VAR103	INVALID	0.200	✓	4.00
VAR155	INVALID	0.254	✓	4.26

Chapter 6

Related Work

6.1 Intermediate Verification Languages

Several Intermediate Verification Language (IVL) have been developed. A comparison can be seen in Table 6.1, where the main features of interest of five languages and the OOX language are compared.

Table 6.1: A comparison between the key features of different languages.

Language	IVL	Object-oriented	Concurrency
Boogie	X		
CIVL-MS	X		X
Why			
Silver	X		
CIVL-C			X
OOX	X	X	X

Boogie. Boogie is an IVL developed by the RiSE group at Microsoft Research [26]. It can be considered low-level as there is no native support for object-oriented concepts, concurrency or a memory model, but most such things can be modelled. This has the advantage that it is flexible, but comes at the cost of extra development time and design choices for the user. There exist several translations into Boogie, for example Spec# [4] and Java bytecode using BML [9].

Boogie has a simple type system which includes user defined types and the following standard types: (1) booleans; (2) integers (3) bit-vectors of arbitrary size; and (4) a polymorphic map. There is no native support for subtyping, but this can be modelled using partial orders.

The statements that are part of the Boogie language are those that are expected, that is: (1) assignment; (2) while loops; (3) if-then-else statements; (4) control flow statements (goto and return); (5) method call statements; (6) assertions and assumptions; and (7) havoc statements that allow for the assignment of non-deterministic values to variables.

Overall, Boogie is flexible enough to model object-oriented languages as C# and Java, but this flexibility has the downside that low-level aspects need to be modelled explicitly. Boogie has no native support for concurrency.

Microsoft Research also developed a language on top of Boogie: CIVL [21] (note that this language differs from the other similarly language, also named CIVL. For clarity, we will refer to this version of CIVL as CIVL-MS.). It extends Boogie with support for concurrency.

Silver. Silver is an IVL developed at ETH Zürich [28]. It aims to provide an infrastructure to verify permission based logics and separation logic [31] in particular. One mayor different design choice between Silver and most other IVLs is that the heap is defined in the language itself, and need not be explicitly defined, like in Boogie. This choice has been made to accommodate for the permission based logics.

Silver has no explicit language support for object-oriented concepts or concurrency, although flexible enough to model both. The downside of this approach is that the every possible interleaving of the target language has to be modelled and thus optimizations such as partial order reduction are not applied.

The back-end of Silver, named Viper, consists of two verification tools. The first one is a verification condition generator, which generates formulas which in turn can be passed to a SMT solver. The second one is based on symbolic execution, which reasons about heap properties directly and resorts for non-heap related properties to a SMT solver.

Why. Why is a tool for deductive program verification [17]. It specifies a programming and specification language called WhyML. It is used as an intermediate verification language to verify C, Java and Ada programs.

WhyML is a dialect of ML, with several modifications. Nested functions and partial function application are supported, but higher-order functions are not. WhyML does not make explicit use of a memory model, but instead statically determines the possible aliases. WhyML does not support concurrency.

CIVL-C. CIVL-C (not to be confused with CIVL from Microsoft Research) is an IVL that is a superset of ANSI-C with a formal semantics [32]. It adds constructs to simplify the modelling of concurrency, verification and memory aspects of the original C language. CIVL-C includes a verification back-end, named CIVL. For more details about CIVL, see Section 6.2.

6.2 Formal Software Verification

Formal software verification is a widely studied field where two approaches are common practice: symbolic execution and model checking. There exists a variety of mature tools with different supported language constructs and target languages. A selection can be seen in Table 6.2.

Table 6.2: A selection of tools for program verification.

Name	Approach	Concurrency	Target languages
KLEE	Symbolic execution		LLVM bitcode
Cloud9	Symbolic execution (cloud)		LLVM bitcode
VerCors	Symbolic execution	✓	Java, OpenCL, PVL
CBMC/JBMC	Symbolic model checking		C, C++ / Java bytecode
CIVL	Symbolic execution	✓	CIVL-C

KLEE and Cloud9. KLEE and Cloud9 are tools based on symbolic execution to automatically generate tests aiming at high test coverage and verify programs in LLVM bitcode [8]. Several optimizations are built into KLEE:

1. Compact State Representation: a representation of the heap as an immutable map which allows for the sharing between different states;
2. Query Optimizations: (1) expression rewriting; (2) implied value concretization, for example when a constraint has the form $x + 1 = 10$, $x = 9$ can be deduced; and (3) a counter-example cache; and
3. State Scheduling: different techniques to explore the search space, e.g. random path selection and targeting uncovered code.

Cloud9 is a tool based on KLEE and parallelizes the symbolic execution engine to achieve a speed-up. Secondly, Cloud9 runs in the cloud, like Amazon EC2 [10].

VerCors. VerCors is a tool developed by the Formal Methods and Tools group at the University of Twente. It aims to verify concurrent programs, written in either Java, C, OpenCL, OpenMP and its own prototypical verification language PVL [6]. VerCors allows for the verification of the absence of data-races, memory safety and functional correctness. It uses Concurrent Separation Logic (CSL) as its underlying logic [30].

Concurrent Separation Logic, an extension of Hoare logic, is a specialized logic to reason about properties of the heap in a concurrent setting. Two main concepts of CSL are the notions of ownership and disjointness. The ownership of a heap property must be made explicit. The notion of disjointness, described using the operator $P * Q$, denotes that P and Q are disjoint. They are not allowed to both write to the same location in the heap [30].

CBMC and JBMC. CBMC and JBMC are tools based on (bounded) symbolic model checking, both using the CPROVER back-end [11]. CBMC targets C and C++ where JBMC targets Java bytecode. CBMC supports verifying pointer safety, array bounds and user-provided assertions. CBMC unwinds and expands all loops, function calls and goto statement until a feasible depth is reached. This feasibility is verified by generating unwinding assertions, which verify that enough unwinding has been done. CBMC replaces all assignments such that they have a static-single assignment form, removing all side-effects from the original program. CBMC generates formulas in conjunctive normal form, which are verified using a SMT solver.

JBMC [12] is an extension of CBMC, adding support for Java. JBMC is a front-end for CBMC, which translates Java bytecode into a CFG representation, suitable for CBMC. JBMC includes an abstract representation of the standard Java libraries, which they call the operation model.

At this point CBMC and JBMC do not have support for concurrency.

CIVL. CIVL is the verification back-end of the IVL CIVL-C [32], as described in Section 6.1. It is based on symbolic execution and is able to find, among other things, assertion violations, deadlocks and memory leaks. Instead of directly using a SMT solver, it uses the Symbolic Algebra and Reasoning Library, which reasons about the formulas. If this library cannot give a definitive answer, it invokes a SMT solver to give a conclusive result. CIVL uses a logical model of the memory, in contrast to a physical memory model, which OOX uses.

CIVL has support for concurrency and optimizes the symbolic execution with partial order reduction. They describe a generalized version of [14].

Chapter 7

Conclusions and Future Work

We presented the intermediate verification language, OOX, with a formally defined static- and dynamic syntax, presented our symbolic execution engine for OOX and shown the results of our experiments.

We have shown that OOX is capable of modelling several algorithms, concurrency and data structures. OOX currently lacks the convenience of expressive expressions to be used in specifications, assumptions and assertions, although this problem can be mitigated by integrating the specifications with the algorithm or logic of the code itself.

We first evaluated the completeness of our Symbolic Execution Engine, the soundness and completeness of the optimizations and the efficacy of the optimizations. For the first experiment, we have shown that the SEE found the bugs in all mutants that were invalid according to its specification and terminated. The same holds true for the second experiment. We have shown that our approach is also able to find the deadlock in the fourth experiment. Suggesting that our approach is complete. The optimizations show to have a drastic effect in most cases, being able to reduce the runtime from 100 seconds to 10 seconds in some extremes. We expected the random interleaving exploration to have a positive effect on the runtime, but the experiments suggest that this is not the case. Random interleaving exploration seems to double the runtime on average for the dining philosophers problem. The verification results of all experiments in all different configurations yielded were equal, suggesting that the optimizations are sound and complete.

We evaluated how scalable our Symbolic Execution Engine is. It has been shown that our SEE is able to completely explore the search space of the algorithm for finding the minimum element in a linked list, up to a linked list of size 5 with maximum depth 120, which is well above necessary, in reasonable time. Partial Order Reduction is shown to be a necessity when verifying concurrent programs, as expected.

Finally, to verify the soundness of our Symbolic Execution Engine, we compared it with CBMC and CIVL, two established tools for which it is reasonable to assume soundness. All experiments showed that the verification results were equal. The runtimes show that our SEE is on par with CBMC and CIVL. There was one outlier in the concurrent mergesort comparison experiment, the mutant CMP2, for which our SEE needed 130 seconds, while CIVL managed to verify the same mutant in 4 seconds. Further investigation of this outlier can be helpful to improve our SEE.

7.1 Future Work

This thesis provides a basis for possible future work. We provide several of such directions.

Extending the OOX Language. It is thinkable that most real-world Java and C# programs consists of more complex constructs which are not part of this thesis. Extending the OOX language with support to model such constructs would be beneficial. One possibility is to extend OOX with user-defined subtyping, allowing for more Java and C# programs to be modelled using OOX. This will lead to new challenges in the symbolic execution engine as, for example, dynamic dispatch might be required.

Another interesting direction is to extend the specification language to be more expressive. For example, extending the specifications to be metamorphic. Research on this subject has already been done, of which the work by Barnett et al. [5] is a promising starting point.

Exploring Different Partial Order Approaches. The partial order reduction approach resulted in a massive reduction of runtime. The partial order approach taken has the advantage that it fits neatly with symbolic execution, unlike many other partial order approaches, which are more suitable for application in model checking. The current technique can be improved by, instead of defining the independence relation only on the reference level, defining it on the reference, field and array index level. Another way to optimize is to identify parts of the symbolic execution engine where one can decrease the possible concrete references that a symbolic reference can point to.

Extensive research has been done on partial order reduction, some claiming to be optimal. The improved variant [23] of the approach we used claims to be optimal for an arbitrary number of threads. This is worthwhile to investigate, as the experiments showed that partial order reduction can drastically improve the scalability of the symbolic execution.

Exploring Other Optimization Techniques. The literature contains a vast collection of other techniques aiming to reduce the effects of the path explosion problem. The survey by Baldoni et al. [3] presents a wide variety of optimization techniques, other than described in this thesis. Other techniques can be explored as well, like the use of program slicing [16, 33].

Bibliography

[1] IEEE Draft Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition). *IEEE P1490/D1, May 2011*, pages 1–505, June 2011.

[2] Stephen Adams. Implementing sets efficiently in a functional language. 1992.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono DâĂŹelia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[5] Mike Barnett, David A Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on formal techniques for Java-like programs (FTfJP)*, 2004.

[6] Stefan Blom and Marieke Huisman. The vercors tool for verification of concurrent programs. In *International Symposium on Formal Methods*, pages 127–131. Springer, 2014.

[7] Jeremy S Bradbury, James R Cordy, and Juergen Dingel. Mutation operators for concurrent java (j2se 5.0). In *Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)*, pages 11–11. IEEE, 2006.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] Jacek Chrzaszcz, Marieke Huisman, and Aleksy Schubert. Bml and related tools. In *International Symposium on Formal Methods for Components and Objects*, pages 278–297. Springer, 2008.

[10] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[12] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. Jbmc: A bounded model checking tool for verifying java bytecode. In *International Conference on Computer Aided Verification*, pages 183–190. Springer, 2018.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] Matthew B Dwyer, John Hatcliff, Venkatesh Prasad Ranganath, et al. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004.

[15] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[16] RR Eilers. Fine-grained model slicing for faster verification. Master's thesis, 2018.

[17] Jean-Christophe Filliâtre and Andrei Paskevich. Why3âĂŤwhere programs meet provers. In *European Symposium on Programming*, pages 125–128. Springer, 2013.

[18] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, volume 40, pages 110–121. ACM, 2005.

[19] Simson Garfinkel. History's worst software bugs. *Wired News, Nov*, 2005.

[20] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 854–865. ACM, 2015.

[21] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *International Conference on Computer Aided Verification*, pages 449–465. Springer, 2015.

[22] Robert Husák and Filip Zavoral. Source code assertion verification using backward symbolic execution. In *AIP Conference Proceedings*, volume 2116, page 350004. AIP Publishing LLC, 2019.

[23] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413. Springer, 2009.

[24] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[25] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[26] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[27] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011.

[28] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.

[29] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[30] Wytse Hendrikus Marinus Oortwijn. *Deductive techniques for model-based concurrency verification*. PhD thesis, 2019.

[31] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[32] Stephen F Siegel, Manchun Zheng, Ziqing Luo, Timothy K Zirkel, Andre V Marianiello, John G Edenhofner, Matthew B Dwyer, and Michael S Rogers. Civl: the concurrency intermediate verification language. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[33] Josep Silva. A vocabulary of program slicing-based techniques. *ACM computing surveys (CSUR)*, 44(3):1–41, 2012.

[34] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 382–396. Springer, 2008.

[35] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M Kirby. Efficient stateful dynamic partial order reduction. In *International SPIN Workshop on Model Checking of Software*, pages 288–305. Springer, 2008.

Appendix A

The Concrete Syntax of the OOX language

A.1 Lexical Structure

⟨*input*⟩ ::= (⟨*section*⟩* ⟨*"newline character"*⟩)*

⟨*section*⟩ ::= ⟨*"whitespace"*⟩
| ⟨*comment*⟩
| ⟨*token*⟩

⟨*comment*⟩ ::= \\ ⟨*"any character except newline characters"*⟩

⟨*token*⟩ ::= ⟨*identifier*⟩
| ⟨*keyword*⟩
| ⟨*literal*⟩
| ⟨*punctuator*⟩

⟨*identifier*⟩ ::= ⟨*"any english alphabet character"*⟩+

⟨*keyword*⟩ ::=
| `assert` | `assume` | `bool` | `break`
| `catch` | `char` | `class` | `continue`
| `else` | `ensures` | `exceptional` | `exists`
| `false` | `float` | `forall` | `fork`
| `if` | `int` | `join` | `lock`
| `new` | `null` | `requires` | `return`
| `static` | `string` | `this` | `throw`
| `true` | `try` | `uint` | `void`
| `while`

⟨*literal*⟩ ::= ⟨*boolean*⟩
| ⟨*integer*⟩
| ⟨*float*⟩
| ⟨*char*⟩
| ⟨*string*⟩
| `null`

⟨*boolean*⟩ ::= `true` | `false`

⟨*integer*⟩ ::= ⟨*digit*⟩+

⟨*float*⟩ ::= ⟨*digit*⟩+ `.` ⟨*digit+*⟩

⟨*char*⟩ ::= `'` ⟨*"any character except ' and newline characters"*⟩ `'`

⟨*string*⟩ ::= `"` ⟨*"any character except " and newline characters"*⟩ `"`

⟨*punctuator*⟩ ::= `+` | `-` | `*` | `/` | `%` | `!` | `:=` | `<`
| `>` | `<=` | `>=` | `==` | `!=` | `&&` | `||` | `==>`
| `{` | `}` | `[` | `]` | `(` | `)` | `.` | `,`
| `;` | `:` | `#`

⟨*digit*⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

A.2 Syntactical Structure

⟨*program*⟩ ::= ⟨*class*⟩*

⟨*class*⟩ ::= `class` ⟨*identifier*⟩ `{` ⟨*member*⟩* `}`

⟨*member*⟩ ::= ⟨*constructor*⟩ | ⟨*method*⟩ | ⟨ *field*⟩

⟨*constructor*⟩ ::= ⟨*identifier*⟩ `(` ⟨*parameters*⟩ `)` ⟨*specification*⟩ ⟨*body*⟩

⟨*method*⟩ ::= `static`? ⟨*type*⟩ ⟨*identifier*⟩ `(` ⟨*parameters*⟩ `)` ⟨*specification*⟩ ⟨*body*⟩

⟨*parameters*⟩ ::= ⟨*parameter*⟩ (`,` ⟨*parameter*⟩)* | ϵ

⟨*parameter*⟩ ::= ⟨*nonvoidtype*⟩ ⟨*identifier*⟩

⟨*specification*⟩ ::= `requires (` ⟨*verificationexpression*⟩ `)`
`ensures (` ⟨*verificationexpression*⟩ `)`
`exceptional (` ⟨*verificationexpression*⟩ `)`

⟨*field*⟩ ::= ⟨*nonvoidtype*⟩ ⟨*identifier*⟩ `;`

⟨*body*⟩ ::= `{` ⟨*statement*⟩+ `}`

⟨*type*⟩ ::= `void` | ⟨*nonvoidtype*⟩

⟨*nonvoidtype*⟩ ::= ⟨*primitivetype*⟩ | ⟨*referencetype*⟩

⟨*primitivetype*⟩ ::= `uint` | `int` | `bool` | `float` | `char`

⟨*referencetype*⟩ ::= ⟨*classtype*⟩ | ⟨*arraytype*⟩

⟨*classtype*⟩ ::= ⟨*identifier*⟩ | `string`

⟨*arraytype*⟩ ::= (⟨*classtype*⟩ | ⟨*primitivetype*⟩) (`[ ]`)+

⟨*statements*⟩ ::= ⟨*statement*⟩*

⟨*statement*⟩ ::= ⟨*declaration*⟩
| ⟨*assignment*⟩
| ⟨*call*⟩
| ⟨*skip*⟩
| ⟨*assert*⟩
| ⟨*assume*⟩
| ⟨*while*⟩
| ⟨*ite*⟩
| ⟨*continue*⟩
| ⟨*break*⟩
| ⟨*return*⟩
| ⟨*throw*⟩
| ⟨*try*⟩
| ⟨*block*⟩
| ⟨*lock*⟩
| ⟨*fork*⟩
| ⟨*join*⟩

⟨*declaration*⟩ ::= ⟨*nonvoidtype*⟩ ⟨*identifier*⟩ `;`

⟨*assignment*⟩ ::= ⟨*lhs*⟩ `:=` ⟨*rhs*⟩ `;`

⟨*lhs*⟩ ::= ⟨*identifier*⟩
| ⟨*identifier*⟩ `.` ⟨*identifier*⟩
| ⟨*identifier*⟩ `[` ⟨*expression*⟩ `]`

⟨*rhs*⟩ ::= ⟨*expression*⟩
| ⟨*identifier*⟩ `.` ⟨*identifier*⟩
| ⟨*invocation*⟩
| ⟨*identifier*⟩ `[` ⟨*expression*⟩ `]`
| `new` ⟨*identifier*⟩ `(` ⟨*arguments*⟩ `)`
| `new` (⟨*classtype*⟩ | ⟨*primitivetype*⟩) (`[` ⟨*integer*⟩ `]`)+

⟨*call*⟩ ::= ⟨*invocation*⟩ `;`

⟨*invocation*⟩ ::= ⟨*identifier*⟩ `.` ⟨*identifier*⟩ `(` ⟨*arguments*⟩ `)`

⟨*arguments*⟩ ::= ⟨*expression*⟩ (`,` ⟨*expression*⟩)* | ϵ

⟨*skip*⟩ ::= `;`

⟨*assert*⟩ ::= `assert` ⟨*verificationexpression*⟩ `;`

⟨*assume*⟩ ::= `assume` ⟨*verificationexpression*⟩ `;`

⟨*while*⟩ ::= `while (` ⟨*expression*⟩ `)` ⟨*statement*⟩

⟨*ite*⟩ ::= `if (` ⟨*expression*⟩ `)` ⟨*statement*⟩ `else` ⟨*statement*⟩

⟨*continue*⟩ ::= `continue ;`

⟨*break*⟩ ::= `break ;`

⟨*return*⟩ ::= `return` ⟨*expression*⟩? `;`

⟨*throw*⟩ ::= `throw ;`

⟨*try*⟩ ::= `try {` ⟨*statements*⟩ `} catch {` ⟨*statements*⟩ `}`

⟨*block*⟩ ::= `{` ⟨*statements*⟩ `}`

⟨*lock*⟩ ::= `lock (` ⟨*identifier*⟩ `) {` ⟨*statements*⟩ `}`

⟨*fork*⟩ ::= `fork` ⟨*invocation*⟩ `;`

⟨*join*⟩ ::= `join ;`

⟨*expression*⟩ ::= ⟨*expression2*⟩

⟨*verificationexpression*⟩ ::= ⟨*expression1*⟩

⟨*expression1*⟩ ::= `forall` ⟨*identifier*⟩ `,` ⟨*identifier*⟩ `:` ⟨*identifier*⟩ `:` ⟨*expression1*⟩
| `exists` ⟨*identifier*⟩ `,` ⟨*identifier*⟩ `:` ⟨*identifier*⟩ `:` ⟨*expression1*⟩
| ⟨*expression2*⟩

⟨*expression2*⟩ ::= ⟨*expression3*⟩ `==>` ⟨*expression2*⟩
| ⟨*expression3*⟩

⟨*expression3*⟩ ::= ⟨*expression4*⟩ `&&` ⟨*expression3*⟩
| ⟨*expression4*⟩ `||` ⟨*expression3*⟩
| ⟨*expression4*⟩

⟨*expression4*⟩ ::= ⟨*expression5*⟩ `==` ⟨*expression4*⟩
| ⟨*expression5*⟩ `!=` ⟨*expression4*⟩
| ⟨*expression5*⟩

⟨*expression5*⟩ ::= ⟨*expression6*⟩ `<` ⟨*expression5*⟩
| ⟨*expression6*⟩ `>` ⟨*expression5*⟩
| ⟨*expression6*⟩ `<=` ⟨*expression5*⟩
| ⟨*expression6*⟩ `>=` ⟨*expression5*⟩
| ⟨*expression6*⟩

⟨*expression6*⟩ ::= ⟨*expression7*⟩ `+` ⟨*expression6*⟩
| ⟨*expression7*⟩ `-` ⟨*expression6*⟩
| ⟨*expression7*⟩

⟨*expression7*⟩ ::= ⟨*expression8*⟩ `*` ⟨*expression7*⟩
| ⟨*expression8*⟩ `/` ⟨*expression7*⟩
| ⟨*expression8*⟩ `%` ⟨*expression7*⟩
| ⟨*expression8*⟩

⟨*expression8*⟩ ::= `-` ⟨*expression8*⟩
| `!` ⟨*expression8*⟩
| ⟨*expression9*⟩

⟨*expression9*⟩ ::= ⟨*identifier*⟩
| `#` ⟨*identifier*⟩
| `(` ⟨*expression*⟩ `)`
| ⟨*literal*⟩

Appendix B

Experimental Data

B.1 Bubblesort

Name	Result	SC (s)	S (s)	C (s)	N (s)
ORIGINAL	VALID	0.342	0.644	4.888	10.600
ARITH5	VALID	0.315	0.538	4.457	10.504
ARITH6	VALID	0.320	0.541	4.396	10.449
ARITH7	VALID	0.328	0.557	4.582	10.351
ARITH9	VALID	0.456	0.859	6.774	13.607
ARITH10	VALID	0.429	0.812	6.113	11.524
CMP0	VALID	0.350	0.536	4.536	10.389
DEL0	VALID	0.337	0.538	4.494	10.418
DEL1	VALID	0.432	0.848	4.391	10.606
DEL5	VALID	0.337	0.526	4.654	10.825
DEL6	VALID	0.344	0.514	4.558	10.781
DEL8	VALID	0.476	0.872	4.358	13.801
LIT1	VALID	0.438	0.844	4.451	10.386
LIT6	VALID	0.345	0.583	4.682	10.533
LIT9	VALID	0.467	0.859	6.753	13.528
VAR8	VALID	0.330	0.550	4.613	10.432
VAR9	VALID	0.338	0.535	4.582	10.496
VAR12	VALID	0.322	0.539	4.483	10.416
VAR13	VALID	0.354	0.558	4.524	10.481
VAR18	VALID	0.469	0.854	4.367	13.397
VAR19	VALID	0.449	0.860	4.337	13.508
ARITH1	INVALID (ensures)	0.205	0.244	0.993	1.662
ARITH2	INVALID (ensures)	0.178	0.248	1.007	1.670
ARITH3	INVALID (ensures)	0.327	0.559	2.304	3.148
CMP1	INVALID (ensures)	0.180	0.222	0.402	0.440
CMP2	INVALID (ensures)	0.176	0.231	0.425	0.424
CMP4	INVALID (ensures)	0.104	0.098	3.523	6.973
DEL3	INVALID (ensures)	0.169	0.201	2.940	4.829

DEL4	INVALID (ensures)	0.153	0.162	1.257	2.524
DEL7	INVALID (ensures)	0.218	0.299	1.281	1.909
LIT0	INVALID (ensures)	0.048	0.057	4.455	10.619
LIT10	INVALID (ensures)	0.262	0.401	1.336	1.972
LIT3	INVALID (ensures)	0.162	0.236	0.431	0.483
LIT4	INVALID (ensures)	0.168	0.243	1.028	1.675
LIT8	INVALID (ensures)	0.232	0.303	1.283	1.950
UN0	INVALID (ensures)	0.054	0.064	4.546	10.320
VAR1	INVALID (ensures)	0.175	0.233	2.002	3.332
VAR2	INVALID (ensures)	0.184	0.258	0.986	1.635
VAR14	INVALID (ensures)	0.189	0.249	1.030	1.634
VAR16	INVALID (ensures)	0.209	0.301	1.194	1.707
VAR15	INVALID (ensures)	0.211	0.247	1.047	1.682
VAR17	INVALID (ensures)	0.188	0.253	1.028	1.632
ARITH0	INVALID (exceptional)	0.130	0.124	0.168	0.173
ARITH4	INVALID (exceptional)	0.178	0.149	0.208	0.233
ARITH8	INVALID (exceptional)	0.155	0.182	0.259	0.281
ARITH11	INVALID (exceptional)	0.143	0.192	0.260	0.298
CMP3	INVALID (exceptional)	0.174	0.236	0.176	0.195
CMP5	INVALID (exceptional)	0.120	0.290	0.281	0.272
DEL2	INVALID (exceptional)	0.113	0.124	0.149	0.141
LIT2	INVALID (exceptional)	0.122	0.124	0.132	0.163
LIT5	INVALID (exceptional)	0.116	0.122	0.182	0.163
LIT7	INVALID (exceptional)	0.143	0.143	0.223	0.203
VAR0	INVALID (exceptional)	0.116	0.130	0.186	0.183
VAR3	INVALID (exceptional)	0.140	0.139	0.201	0.179
VAR4	INVALID (exceptional)	0.120	0.131	0.169	0.175
VAR5	INVALID (exceptional)	0.267	0.244	0.199	0.193
VAR6	INVALID (exceptional)	0.136	0.146	0.193	0.186
VAR7	INVALID (exceptional)	0.141	0.134	0.189	0.203
VAR10	INVALID (exceptional)	0.136	0.162	0.211	0.217
VAR11	INVALID (exceptional)	0.129	0.150	0.234	0.230
VAR20	INVALID (exceptional)	0.162	0.161	0.253	0.238
VAR21	INVALID (exceptional)	0.158	0.161	0.248	0.234

B.2 Minimum element in Linked List

Name	**Result**	**SC (s)**	**S (s)**	**C (s)**	**N (s)**
ORIGINAL	VALID	8.125	7.797	76.218	76.728
ARITH3	VALID	61.242	58.983	80.835	80.844
ARITH4	VALID	29.939	30.531	52.165	53.957
ARITH5	VALID	29.700	29.613	51.703	51.971
CMP2	VALID	7.714	7.736	77.599	77.512
DEL4	VALID	8.166	8.164	79.384	79.352
DEL5	VALID	7.746	7.767	103.350	103.098

DEL7	VALID	1.864	2.145	66.456	106.410
DEL9	VALID	0.628	0.701	2.265	2.686
DEL11	VALID	56.278	56.503	79.915	82.629
LIT6	VALID	58.517	57.949	78.601	78.659
VAR9	VALID	5.851	5.847	75.989	76.761
VAR10	VALID	4.209	7.573	24.306	70.866
VAR23	VALID	0.718	1.003	2.084	2.692
VAR24	VALID	0.902	1.035	2.524	2.876
VAR30	VALID	7.886	7.885	77.293	77.752
VAR32	VALID	54.572	54.673	75.995	74.647
VAR33	VALID	57.309	56.630	76.116	75.924
VAR37	VALID	7.172	7.188	55.967	56.099
CMP3	INVALID (ensures)	1.195	1.203	10.322	10.205
CMP4	INVALID (ensures)	1.177	1.183	9.939	9.832
DEL6	INVALID (ensures)	0.076	0.063	0.107	0.097
DEL8	INVALID (ensures)	1.117	1.105	9.561	9.479
DEL10	INVALID (ensures)	1.174	1.148	76.753	75.652
EQ2	INVALID (ensures)	7.968	8.054	78.971	78.509
LIT5	INVALID (ensures)	0.352	0.311	3.040	3.010
LIT7	INVALID (ensures)	0.337	0.357	3.116	3.078
VAR5	INVALID (ensures)	0.097	0.081	0.123	0.113
VAR13	INVALID (ensures)	1.020	1.056	9.024	9.074
VAR15	INVALID (ensures)	1.020	1.006	8.953	8.972
VAR16	INVALID (ensures)	1.076	1.048	9.371	9.305
VAR17	INVALID (ensures)	0.977	1.011	8.687	8.641
VAR18	INVALID (ensures)	1.028	1.024	8.727	8.645
VAR19	INVALID (ensures)	7.218	7.227	72.054	70.539
VAR21	INVALID (ensures)	1.025	1.002	8.963	9.012
VAR22	INVALID (ensures)	7.582	7.575	79.089	80.521
VAR29	INVALID (ensures)	1.157	1.117	9.721	9.701
ARITH2	INVALID (exceptional)	0.367	0.317	0.495	0.445
DEL3	INVALID (exceptional)	0.152	0.135	0.149	0.151
LIT4	INVALID (exceptional)	0.150	0.137	0.169	0.181
VAR36	INVALID (exceptional)	0.326	0.320	0.469	0.463
VAR4	INVALID (exceptional)	0.157	0.131	0.165	0.161
VAR6	INVALID (exceptional)	0.068	0.067	0.080	0.087
VAR8	INVALID (exceptional)	0.130	0.136	0.160	0.158
VAR12	INVALID (exceptional)	0.177	0.149	0.182	0.178
VAR26	INVALID (exceptional)	0.215	0.132	0.187	0.164
VAR27	INVALID (exceptional)	0.143	0.132	0.181	0.180
VAR35	INVALID (exceptional)	0.369	0.358	0.500	0.507

B.3 Concurrent Mergesort

Name	**Result**	**A/rand (s)**	**A/def (s)**

ORIGINAL	VALID	1.163	1.194
ARITH4	VALID	37.034	37.648
ARITH5	VALID	18.708	18.666
ARITH6	VALID	37.258	37.191
ARITH7	VALID	11.311	11.411
ARITH8	VALID	19.977	19.956
ARITH9	VALID	0.436	0.428
ARITH10	VALID	10.326	10.348
ARITH11	VALID	10.498	10.288
ARITH12	VALID	178.864	178.424
ARITH13	VALID	25.063	24.940
ARITH14	VALID	25.525	25.005
ARITH15	VALID	122.039	122.673
ARITH20	VALID	1.150	1.147
ARITH22	VALID	0.319	0.319
ARITH23	VALID	0.319	0.316
ARITH29	VALID	1.164	1.187
ARITH30	VALID	1.168	1.138
ARITH31	VALID	1.165	1.234
ARITH33	VALID	1.404	1.444
ARITH34	VALID	1.350	1.333
ARITH35	VALID	1.342	1.337
ARITH37	VALID	1.198	1.178
ARITH38	VALID	1.206	1.202
ARITH39	VALID	1.170	1.145
ARITH41	VALID	1.475	1.333
ARITH42	VALID	1.314	1.330
ARITH44	VALID	1.140	1.125
ARITH45	VALID	1.136	1.142
ARITH46	VALID	1.147	1.139
ARITH47	VALID	1.184	1.313
ARITH52	VALID	1.140	1.140
ARITH53	VALID	1.157	1.133
ARITH54	VALID	1.128	1.142
ARITH55	VALID	1.234	1.224
ARITH60	VALID	1.191	1.147
ARITH61	VALID	1.213	1.102
ARITH62	VALID	1.004	0.999
ARITH63	VALID	1.012	1.006
ARITH65	VALID	1.166	1.147
ARITH66	VALID	1.150	1.135
ARITH67	VALID	1.144	1.166
CMP0	VALID	42.602	42.966
CMP3	VALID	1.178	1.200
CMP4	VALID	1.157	1.177
CMP5	VALID	1.174	1.164

CMP18	VALID	1.160	1.165
DEL2	VALID	18.910	18.546
DEL3	VALID	0.446	0.453
DEL4	VALID	0.544	0.634
DEL5	VALID	80.056	79.138
DEL8	VALID	1.156	1.172
DEL10	VALID	1.294	1.138
DEL14	VALID	1.164	1.147
DEL15	VALID	1.396	1.363
DEL17	VALID	1.167	1.179
DEL18	VALID	1.381	1.360
DEL20	VALID	1.154	1.136
DEL23	VALID	1.169	1.157
DEL25	VALID	1.147	1.141
DEL27	VALID	1.179	1.145
FORK0	VALID	0.355	0.344
LIT6	VALID	21.951	22.597
LIT7	VALID	0.426	0.430
LIT8	VALID	25.257	24.696
LIT9	VALID	1.173	1.156
LIT11	VALID	1.152	1.153
LIT16	VALID	1.157	1.149
LIT18	VALID	1.372	1.372
LIT19	VALID	1.159	1.164
LIT20	VALID	1.207	1.146
LIT22	VALID	1.376	1.393
LIT23	VALID	1.142	1.145
LIT24	VALID	1.136	1.132
LIT25	VALID	1.189	1.149
LIT27	VALID	1.186	1.170
LIT28	VALID	1.181	1.180
LIT29	VALID	1.185	1.164
LIT31	VALID	1.170	1.175
LIT32	VALID	1.164	1.134
LIT33	VALID	1.146	1.183
VAR0	VALID	10.232	10.289
VAR2	VALID	1.174	1.157
VAR3	VALID	22.114	22.225
VAR4	VALID	0.428	0.415
VAR5	VALID	10.210	10.201
VAR6	VALID	0.431	0.415
VAR7	VALID	0.451	0.425
VAR8	VALID	0.428	0.418
VAR9	VALID	22.128	22.207
VAR10	VALID	4.923	5.020
VAR11	VALID	1.147	1.143

VAR12	VALID	1.150	1.130
VAR13	VALID	1.172	1.154
VAR14	VALID	1.151	1.143
VAR16	VALID	1.133	1.142
VAR25	VALID	1.175	1.163
VAR27	VALID	1.116	1.151
VAR28	VALID	1.160	1.148
VAR30	VALID	1.179	1.168
VAR31	VALID	1.170	1.149
VAR37	VALID	1.157	1.170
VAR38	VALID	1.124	1.139
VAR39	VALID	1.186	1.292
VAR41	VALID	1.143	1.143
VAR43	VALID	1.147	1.205
VAR44	VALID	1.164	1.145
VAR52	VALID	1.168	1.166
VAR53	VALID	1.167	1.160
VAR54	VALID	1.159	1.160
VAR55	VALID	1.145	1.136
VAR56	VALID	1.282	1.118
VAR57	VALID	1.170	1.151
VAR58	VALID	1.199	1.255
VAR59	VALID	1.187	1.155
VAR66	VALID	1.162	1.143
VAR68	VALID	1.149	1.142
VAR69	VALID	1.155	1.173
VAR70	VALID	1.141	1.141
VAR71	VALID	1.181	1.180
VAR72	VALID	1.141	1.148
VAR73	VALID	1.164	1.151
VAR77	VALID	1.164	1.146
VAR78	VALID	1.147	1.133
VAR79	VALID	1.177	1.179
VAR80	VALID	1.190	1.168
VAR81	VALID	1.165	1.177
VAR85	VALID	1.144	1.155
VAR86	VALID	1.179	1.179
VAR87	VALID	1.166	1.151
VAR88	VALID	1.200	1.143
VAR89	VALID	1.192	1.184
VAR90	VALID	1.154	1.145
VAR91	VALID	1.186	1.175
VAR92	VALID	1.157	1.164
VAR93	VALID	1.180	1.218
VAR94	VALID	1.208	1.157
VAR95	VALID	1.362	1.331

VAR96	VALID	1.167	1.156
VAR99	VALID	1.180	1.154
VAR101	VALID	1.154	1.146
VAR102	VALID	1.217	1.172
VAR104	VALID	1.378	1.351
VAR105	VALID	1.422	1.365
VAR106	VALID	1.385	1.374
VAR107	VALID	1.348	1.338
VAR108	VALID	1.377	1.344
VAR109	VALID	1.367	1.358
VAR110	VALID	1.358	1.377
VAR113	VALID	1.155	1.147
VAR114	VALID	1.169	1.148
VAR115	VALID	1.161	1.154
VAR116	VALID	1.173	1.174
VAR117	VALID	1.128	1.131
VAR121	VALID	1.149	1.141
VAR122	VALID	1.153	1.140
VAR123	VALID	1.156	1.146
VAR124	VALID	1.159	1.137
VAR125	VALID	1.154	1.166
VAR129	VALID	1.176	1.158
VAR130	VALID	1.164	1.161
VAR131	VALID	1.273	1.161
VAR132	VALID	1.190	1.162
VAR133	VALID	1.186	1.180
VAR134	VALID	1.170	1.170
VAR135	VALID	1.179	1.155
VAR137	VALID	1.262	1.136
VAR138	VALID	1.213	1.181
VAR139	VALID	1.153	1.174
VAR140	VALID	1.208	1.178
VAR143	VALID	1.151	1.165
VAR145	VALID	1.165	1.181
VAR146	VALID	1.148	1.146
VAR148	VALID	1.367	1.368
VAR149	VALID	1.362	1.376
VAR150	VALID	1.238	1.127
VAR152	VALID	1.364	1.367
VAR153	VALID	1.409	1.388
VAR154	VALID	1.377	1.379
VAR157	VALID	1.432	1.337
VAR158	VALID	1.232	1.152
VAR159	VALID	1.383	1.379
VAR160	VALID	1.382	1.386
VAR161	VALID	1.167	1.150

VAR162	VALID	1.131	1.152
VAR163	VALID	1.153	1.146
VAR164	VALID	1.168	1.161
VAR165	VALID	1.148	1.145
VAR166	VALID	1.158	1.150
VAR167	VALID	1.291	1.155
VAR168	VALID	1.174	1.157
VAR169	VALID	1.160	1.151
VAR170	VALID	1.144	1.157
VAR171	VALID	1.193	1.160
VAR172	VALID	1.154	1.147
VAR173	VALID	1.146	1.140
VAR174	VALID	1.165	1.161
VAR175	VALID	1.263	1.126
VAR177	VALID	1.137	1.141
VAR184	VALID	1.141	1.156
VAR197	VALID	1.140	1.127
VAR199	VALID	1.144	1.149
VAR200	VALID	1.182	1.153
VAR201	VALID	1.186	1.137
VAR202	VALID	1.192	1.283
VAR204	VALID	1.153	1.155
VAR205	VALID	1.140	1.140
VAR206	VALID	1.189	1.169
VAR207	VALID	1.165	1.139
VAR208	VALID	1.171	1.141
VAR209	VALID	1.156	1.134
VAR213	VALID	1.250	1.295
VAR214	VALID	1.143	1.144
VAR215	VALID	1.154	1.147
VAR216	VALID	1.144	1.143
VAR217	VALID	1.158	1.127
VAR219	VALID	1.176	1.187
VAR223	VALID	1.158	1.176
VAR224	VALID	1.176	1.176
VAR225	VALID	1.143	1.143
VAR226	VALID	1.139	1.118
VAR227	VALID	1.271	1.155
VAR236	VALID	1.154	1.145
VAR239	VALID	1.157	1.132
VAR240	VALID	1.135	1.154
VAR241	VALID	1.137	1.130
VAR242	VALID	1.182	1.186
VAR243	VALID	1.166	1.165
VAR245	VALID	1.236	1.216
VAR246	VALID	1.114	1.146

VAR247	VALID	1.162	1.169
VAR248	VALID	1.305	1.140
VAR249	VALID	1.155	1.159
VAR250	VALID	1.191	1.141
VAR251	VALID	1.147	1.138
VAR252	VALID	1.081	1.071
VAR253	VALID	1.180	1.174
VAR255	VALID	1.165	1.141
VAR256	VALID	1.192	1.164
VAR257	VALID	1.152	1.169
VAR258	VALID	1.155	1.159
VAR259	VALID	1.213	1.189
VAR265	VALID	1.180	1.154
VAR266	VALID	1.185	1.161
VAR269	VALID	1.216	1.252
VAR271	VALID	1.160	1.150
VAR275	VALID	1.154	1.148
VAR280	VALID	1.126	1.119
VAR281	VALID	1.141	1.113
VAR282	VALID	1.147	1.144
VAR283	VALID	1.116	1.115
VAR285	VALID	1.133	1.238
VAR288	VALID	1.174	1.182
VAR289	VALID	1.145	1.148
VAR293	VALID	1.129	1.141
VAR294	VALID	1.165	1.138
VAR295	VALID	1.164	1.153
VAR299	VALID	1.196	1.154
VAR300	VALID	1.152	1.146
VAR301	VALID	1.177	1.146
VAR302	VALID	1.161	1.150
VAR305	VALID	1.161	1.148
VAR307	VALID	1.190	1.181
VAR308	VALID	1.163	1.150
VAR310	VALID	1.183	1.170
VAR311	VALID	1.155	1.152
VAR312	VALID	1.140	1.141
VAR313	VALID	1.150	1.122
VAR314	VALID	1.149	1.148
VAR315	VALID	1.183	1.192
VAR316	VALID	1.168	1.146
VAR319	VALID	1.188	1.162
VAR320	VALID	1.192	1.186
VAR321	VALID	1.174	1.144
VAR322	VALID	1.176	1.159
VAR323	VALID	1.201	1.170

VAR324	VALID	1.206	1.151
VAR325	VALID	1.125	1.126
VAR326	VALID	1.131	1.144
ARITH3	INVALID (ensures)	0.142	0.141
CMP1	INVALID (ensures)	0.152	0.131
CMP2	INVALID (ensures)	130.805	130.765
CMP6	INVALID (ensures)	0.203	0.199
CMP7	INVALID (ensures)	0.198	0.194
CMP9	INVALID (ensures)	0.226	0.204
CMP10	INVALID (ensures)	0.196	0.195
CMP12	INVALID (ensures)	0.259	0.263
CMP15	INVALID (ensures)	0.207	0.232
CMP16	INVALID (ensures)	0.219	0.208
CMP19	INVALID (ensures)	0.258	0.255
CMP20	INVALID (ensures)	0.259	0.249
DEL1	INVALID (ensures)	0.125	0.135
DEL6	INVALID (ensures)	0.165	0.154
DEL11	INVALID (ensures)	0.226	0.211
DEL12	INVALID (ensures)	0.217	0.235
DEL13	INVALID (ensures)	0.230	0.205
DEL16	INVALID (ensures)	0.268	0.251
DEL19	INVALID (ensures)	0.257	0.255
DEL22	INVALID (ensures)	0.219	0.217
DEL26	INVALID (ensures)	0.243	0.259
LIT3	INVALID (ensures)	0.136	0.143
LIT5	INVALID (ensures)	0.149	0.133
LIT13	INVALID (ensures)	0.192	0.208
VAR1	INVALID (ensures)	0.194	0.199
VAR15	INVALID (ensures)	0.205	0.193
VAR17	INVALID (ensures)	0.216	0.196
VAR18	INVALID (ensures)	0.183	0.180
VAR19	INVALID (ensures)	0.203	0.205
VAR20	INVALID (ensures)	0.148	0.144
VAR21	INVALID (ensures)	0.147	0.130
VAR29	INVALID (ensures)	0.188	0.190
VAR32	INVALID (ensures)	0.182	0.195
VAR33	INVALID (ensures)	0.219	0.198
VAR40	INVALID (ensures)	0.214	0.196
VAR45	INVALID (ensures)	0.189	0.193
VAR74	INVALID (ensures)	0.229	0.216
VAR82	INVALID (ensures)	0.222	0.216
VAR118	INVALID (ensures)	0.267	0.263
VAR126	INVALID (ensures)	0.256	0.258
VAR136	INVALID (ensures)	0.250	0.258
VAR176	INVALID (ensures)	0.203	0.198
VAR180	INVALID (ensures)	0.216	0.194

VAR191	INVALID (ensures)	0.195	0.206
VAR193	INVALID (ensures)	0.195	0.202
VAR194	INVALID (ensures)	0.204	0.197
VAR195	INVALID (ensures)	0.211	0.204
VAR196	INVALID (ensures)	0.258	0.265
VAR210	INVALID (ensures)	0.270	0.328
VAR228	INVALID (ensures)	0.268	0.256
VAR229	INVALID (ensures)	0.267	0.262
VAR232	INVALID (ensures)	0.267	0.269
VAR238	INVALID (ensures)	0.229	0.224
VAR244	INVALID (ensures)	0.221	0.215
VAR278	INVALID (ensures)	0.221	0.223
VAR279	INVALID (ensures)	0.214	0.222
VAR284	INVALID (ensures)	0.252	0.271
VAR286	INVALID (ensures)	0.249	0.251
VAR287	INVALID (ensures)	0.249	0.240
VAR290	INVALID (ensures)	0.270	0.293
VAR296	INVALID (ensures)	0.287	0.286
VAR317	INVALID (ensures)	0.254	0.258
VAR318	INVALID (ensures)	0.267	0.255
ARITH0	INVALID (exceptional)	0.346	0.428
ARITH1	INVALID (exceptional)	0.189	0.170
ARITH2	INVALID (exceptional)	0.166	0.176
ARITH16	INVALID (exceptional)	0.145	0.160
ARITH17	INVALID (exceptional)	0.199	0.205
ARITH18	INVALID (exceptional)	0.200	0.205
ARITH19	INVALID (exceptional)	0.198	0.196
ARITH21	INVALID (exceptional)	0.205	0.213
ARITH24	INVALID (exceptional)	0.168	0.173
ARITH25	INVALID (exceptional)	0.209	0.205
ARITH26	INVALID (exceptional)	0.209	0.211
ARITH27	INVALID (exceptional)	0.218	0.214
ARITH28	INVALID (exceptional)	0.207	0.202
ARITH32	INVALID (exceptional)	0.209	0.205
ARITH36	INVALID (exceptional)	0.261	0.252
ARITH40	INVALID (exceptional)	0.270	0.295
ARITH43	INVALID (exceptional)	0.274	0.269
ARITH48	INVALID (exceptional)	0.268	0.247
ARITH49	INVALID (exceptional)	0.268	0.254
ARITH50	INVALID (exceptional)	0.254	0.255
ARITH51	INVALID (exceptional)	0.254	0.249
ARITH56	INVALID (exceptional)	0.208	0.195
ARITH57	INVALID (exceptional)	0.218	0.204
ARITH58	INVALID (exceptional)	0.214	0.209
ARITH59	INVALID (exceptional)	0.211	0.207
ARITH64	INVALID (exceptional)	0.207	0.214

BOOL0	INVALID (exceptional)	0.211	0.244
BOOL1	INVALID (exceptional)	0.233	0.225
CMP8	INVALID (exceptional)	0.230	0.234
CMP11	INVALID (exceptional)	0.238	0.238
CMP13	INVALID (exceptional)	0.212	0.210
CMP14	INVALID (exceptional)	0.214	0.201
CMP17	INVALID (exceptional)	0.217	0.202
DEL7	INVALID (exceptional)	0.181	0.193
DEL9	INVALID (exceptional)	0.217	0.203
DEL21	INVALID (exceptional)	0.253	0.251
DEL24	INVALID (exceptional)	0.216	0.209
LIT0	INVALID (exceptional)	0.129	0.122
LIT1	INVALID (exceptional)	0.135	0.122
LIT2	INVALID (exceptional)	0.161	0.169
LIT4	INVALID (exceptional)	0.168	0.166
LIT10	INVALID (exceptional)	0.193	0.195
LIT12	INVALID (exceptional)	0.216	0.207
LIT14	INVALID (exceptional)	0.204	0.189
LIT15	INVALID (exceptional)	0.202	0.216
LIT17	INVALID (exceptional)	0.215	0.197
LIT21	INVALID (exceptional)	0.267	0.244
LIT26	INVALID (exceptional)	0.242	0.261
LIT30	INVALID (exceptional)	0.202	0.205
VAR22	INVALID (exceptional)	0.169	0.162
VAR23	INVALID (exceptional)	0.197	0.202
VAR24	INVALID (exceptional)	0.197	0.202
VAR26	INVALID (exceptional)	0.347	0.180
VAR34	INVALID (exceptional)	0.204	0.212
VAR35	INVALID (exceptional)	0.232	0.223
VAR36	INVALID (exceptional)	0.208	0.206
VAR42	INVALID (exceptional)	0.211	0.203
VAR46	INVALID (exceptional)	0.192	0.192
VAR47	INVALID (exceptional)	0.182	0.174
VAR48	INVALID (exceptional)	0.190	0.192
VAR49	INVALID (exceptional)	0.236	0.223
VAR50	INVALID (exceptional)	0.246	0.232
VAR51	INVALID (exceptional)	0.239	0.226
VAR60	INVALID (exceptional)	0.190	0.195
VAR61	INVALID (exceptional)	0.216	0.213
VAR62	INVALID (exceptional)	0.195	0.200
VAR63	INVALID (exceptional)	0.248	0.233
VAR64	INVALID (exceptional)	0.343	0.344
VAR65	INVALID (exceptional)	0.233	0.226
VAR67	INVALID (exceptional)	0.164	0.175
VAR75	INVALID (exceptional)	0.194	0.203
VAR76	INVALID (exceptional)	0.194	0.200

VAR83	INVALID (exceptional)	0.189	0.197
VAR84	INVALID (exceptional)	0.205	0.196
VAR97	INVALID (exceptional)	0.219	0.202
VAR98	INVALID (exceptional)	0.209	0.196
VAR100	INVALID (exceptional)	0.191	0.212
VAR103	INVALID (exceptional)	0.205	0.200
VAR111	INVALID (exceptional)	0.207	0.209
VAR112	INVALID (exceptional)	0.216	0.209
VAR119	INVALID (exceptional)	0.235	0.235
VAR120	INVALID (exceptional)	0.236	0.231
VAR127	INVALID (exceptional)	0.240	0.249
VAR128	INVALID (exceptional)	0.241	0.238
VAR141	INVALID (exceptional)	0.259	0.353
VAR142	INVALID (exceptional)	0.257	0.242
VAR144	INVALID (exceptional)	0.235	0.244
VAR147	INVALID (exceptional)	0.251	0.243
VAR151	INVALID (exceptional)	0.273	0.282
VAR155	INVALID (exceptional)	0.400	0.254
VAR156	INVALID (exceptional)	0.268	0.257
VAR178	INVALID (exceptional)	0.230	0.221
VAR179	INVALID (exceptional)	0.220	0.221
VAR181	INVALID (exceptional)	0.225	0.227
VAR182	INVALID (exceptional)	0.225	0.223
VAR183	INVALID (exceptional)	0.216	0.220
VAR185	INVALID (exceptional)	0.216	0.222
VAR186	INVALID (exceptional)	0.241	0.246
VAR187	INVALID (exceptional)	0.252	0.241
VAR188	INVALID (exceptional)	0.249	0.238
VAR189	INVALID (exceptional)	0.239	0.240
VAR190	INVALID (exceptional)	0.246	0.253
VAR192	INVALID (exceptional)	0.273	0.244
VAR198	INVALID (exceptional)	0.254	0.240
VAR203	INVALID (exceptional)	0.243	0.237
VAR211	INVALID (exceptional)	0.260	0.254
VAR212	INVALID (exceptional)	0.252	0.255
VAR218	INVALID (exceptional)	0.252	0.249
VAR220	INVALID (exceptional)	0.255	0.250
VAR221	INVALID (exceptional)	0.247	0.253
VAR222	INVALID (exceptional)	0.254	0.239
VAR230	INVALID (exceptional)	0.221	0.204
VAR231	INVALID (exceptional)	0.196	0.212
VAR233	INVALID (exceptional)	0.209	0.213
VAR234	INVALID (exceptional)	0.210	0.221
VAR235	INVALID (exceptional)	0.198	0.201
VAR237	INVALID (exceptional)	0.212	0.206
VAR254	INVALID (exceptional)	0.193	0.205

VAR260	INVALID (exceptional)	0.207	0.205
VAR261	INVALID (exceptional)	0.195	0.209
VAR262	INVALID (exceptional)	0.200	0.208
VAR263	INVALID (exceptional)	0.216	0.208
VAR264	INVALID (exceptional)	0.209	0.208
VAR267	INVALID (exceptional)	0.218	0.208
VAR268	INVALID (exceptional)	0.208	0.220
VAR270	INVALID (exceptional)	0.209	0.205
VAR272	INVALID (exceptional)	0.209	0.202
VAR273	INVALID (exceptional)	0.199	0.224
VAR274	INVALID (exceptional)	0.206	0.206
VAR276	INVALID (exceptional)	0.217	0.209
VAR277	INVALID (exceptional)	0.204	0.219
VAR291	INVALID (exceptional)	0.222	0.216
VAR292	INVALID (exceptional)	0.205	0.211
VAR297	INVALID (exceptional)	0.198	0.205
VAR298	INVALID (exceptional)	0.205	0.209
VAR303	INVALID (exceptional)	0.227	0.212
VAR304	INVALID (exceptional)	0.207	0.207
VAR306	INVALID (exceptional)	0.201	0.210
VAR309	INVALID (exceptional)	0.211	0.235

B.4 Dining Philosophers

Name	**Result**	**A/rand (s)**	**A/def (s)**
ORIGINAL	DEADLOCK	0.701	0.420

Listing B.1: The bubblesort algorithm in the OOX language.

```
1       static int[] sort(int[] array)
2           requires(array != null)
3           ensures(forall v, i : retval : forall w, j : retval : i < j ==> v <= w)
4           exceptional(false)
5       {
6           bool sorted := false;
7           while (!sorted) {
8               sorted := true;
9               int i := 1;
10              while (i < #array) {
11                  int a := array[i];
12                  int b := array[i - 1];
13                  if (a < b) {
14                      array[i] := b;
15                      array[i - 1] := a;
16                      sorted := false;
17                  }
18                  i := i + 1;
19              }
20          }
21          return array;
22      }
```

Listing B.2: The bubblesort algorithm in C for CBMC.

```
1   void sort(int length)
2   {
3       __CPROVER_assume(length >= 0 && length <= 3);
4       int array[length];
5       for (int i = 0; i < length; i++)
6           array[i] = nondet_int();
7       int sorted = 0;
8       while (!sorted) {
9           sorted = 1;
10          int i = 1;
11          while (i < length) {
12              int a = array[i];
13              int b = array[i-1];
14              if (a < b) {
15                  array[i] = b;
16                  array[i - 1] = a;
17                  sorted = 0;
18              }
19              i = i + 1;
20          }
21      }
22      for (int i = 0; i < length; i++)
23          for (int j = 0; j < length; j++)
24              __CPROVER_assert(!(i < j) || array[i] <= array[j], "postcondition");
25  }
```

Listing B.3: The bubblesort algorithm in CIVL-C.

```
1  $input int length;
2  $assume(length > 0 && length <= 3);
3
4  void sort(int length)
5  {
6      int array[length];
7      $havoc(&array);
8      int sorted = 0;
9      while (!sorted)  {
10         sorted = 1;
11         int i = 1;
12         while (i < length) {
13             int a = array[i];
14             int b = array[i-1];
15             if (a < b) {
16                 array[i] = b;
17                 array[i - 1] = a;
18                 sorted = 0;
19             }
20             i = i + 1;
21         }
22     }
23
24     $assert($forall(int i : 0 .. length - 1)
25                 $forall(int j : 0 .. length - 1) i < j => array[i] <= array[j]);
26 }
27
28 void main()
29 {
30     sort(length);
31 }
```

Listing B.4: The minimum element of a linked list in the OOX language.

```
1  class Node
2  {
3      int value;
4      Node next;
5
6      static int min(Node node)
7          requires(node != null)
8          exceptional(false)
9      {
10         int N := 3;
11         int[] values := new int[N];
12         int i := 0;
13
14         int min := node.value;
15         Node next := node.next;
16         while (next != null && i < N)
17         {
18             int value := next.value;
19             if (value < min) {
20                 min := value;
21             }
22             next := next.next;
23             values[i] := value;
24             i := i + 1;
25         }
26         assert forall value, index : values : min <= value;
27         return min;
28     }
29 }
```

Listing B.5: The concurrent mergesort algorithm in the OOX language.

```
1  static int[] sort(int[] array)
2      requires(array != null)
3      ensures(forall v, i : retval : forall w, j : retval : i < j ==> v <= w)
4      exceptional(false)
5  {
6      Main.mergesort(array, 0, #array - 1);
7      return array;
8  }
9  static void mergesort(int[] array, int left, int right)
10     exceptional(false)
11 {
12     if (left < right) {
13         int middle := (left + right) / 2;
14         fork Main.mergesort(array, left, middle);
15         Main.mergesort(array, middle + 1, right);
16         join;
17         Main.merge(array, left, middle, right);
18     }
19 }
20 static void merge(int[] array, int left, int middle, int right)
21     exceptional(false)
22 {
23     int[] temp := new int[right - left + 1];
24     int i := left;
25     int j := middle + 1;
26     int k := 0;
27     while (i <= middle && j <= right) {
28         int arrayI := array[i];
29         int arrayJ := array[j];
30         if (arrayI <= arrayJ) {
31             temp[k] := array[i];
32             k := k + 1;
33             i := i + 1;
34         }
35         else {
36             temp[k] := array[j];
37             k := k + 1;
38             j := j + 1;
39         }
40     }
41     while (i <= middle) {
42         temp[k] := array[i];
43         k := k + 1;
44         i := i + 1;
45     }
46     while (j <= right) {
47         temp[k] := array[j];
48         k := k + 1;
49         j := j + 1;
50     }
51     i := left;
52     while (i <= right) {
53         array[i] := temp[i - left];
54         i := i + 1;
55     }
56 }
```

Listing B.6: The concurrent mergesort algorithm in CIVL-C.

```
#include <civlc.cvh>
#include <stdlib.h>

$input int length;
$assume(length > 0 && length <= 3);

void merge(int *array, int left, int middle, int right)
{
    int *temp = (int *)malloc((right - left + 1) * sizeof(int));
    int i = left;
    int j = middle + 1;
    int k = 0;
    while (i <= middle && j <= right) {
        int arrayI = array[i];
        int arrayJ = array[j];
        if (arrayI <= arrayJ)
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
        while (i <= middle)
            temp[k++] = array[i++];
        while (j <= right)
            temp[k++] = array[j++];
        for (int i = left; i <= right; i++)
            array[i] = temp[i - left];
    }
}

void mergesort(int *array, int left, int right)
{
    if (left < right) {
        int middle = (left + right) / 2;
        $proc pid = $spawn mergesort(array, left, middle);
        mergesort(array, middle + 1, right);
        $wait(pid);
        merge(array, left, middle, right);
    }
}

void main()
{
    int *array = (int *) malloc(length * sizeof(int));
    mergesort(array, 0, length - 1);
    $assert($forall(int i : 0 .. length - 1)
                $forall(int j : 0 .. length - 1) i < j => array[i] <= array[j]);
}
```

Listing B.7: The dining philosophers problem in the OOX language.

```
1  class Main
2  {
3      static void main()
4          exceptional(false)
5      {
6          int n := 2;
7          Fork[] forks := new Fork[n];
8          int i := 0;
9          while (i < n) {
10             forks[i] := new Fork();
11             i := i + 1;
12         }
13         i := 0;
14         while (i < n) {
15             Fork left := forks[i];
16             Fork right := forks[(i + 1) % n];
17             fork Main.eat(left, right);
18             i := i + 1;
19         }
20         join;
21     }
22     static void eat(Fork left, Fork right)
23     {
24         while (true) {
25             lock (left) {
26                 lock (right) {
27                     ;
28                 }
29             }
30         }
31     }
32 }
33 class Fork {
34     Fork() { ; }
35 }
```