

**A machine checked soundness proof for  
an intermediate verification language:  
extended version**

*Frédéric Vogels    Bart Jacobs    Frank Piessens*

*Report CW 526, October 2008*



**Katholieke Universiteit Leuven**  
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# A machine checked soundness proof for an intermediate verification language: extended version

*Frédéric Vogels    Bart Jacobs    Frank Piessens*

*Report CW 526, October 2008*

Department of Computer Science, K.U.Leuven

## **Abstract**

Machine-checked proofs of properties of programming languages have gained in importance significantly over the past few years. This paper contributes to this trend by proposing an approach for doing machine-checked soundness proofs for verification condition (VC) generators. Our approach embraces the multi-phase VC generation common in modern program verifiers. Such verifiers split the generation of VCs in two (or even more) phases, using an intermediate verification language as the bridge between the programming language and logic. In our approach, we define a formal operational semantics of the intermediate verification language, and we prove the soundness of two translations separately: (1) the translation of the intermediate verification language to VCs, and (2) the translation of the source programming language to the intermediate language. This paper presents a fully machine checked proof of step (1) for a prototypical intermediate verification language, and then illustrates step (2) for a very small object oriented programming language.

# A machine checked soundness proof for an intermediate verification language: extended version

Frédéric Vogels, Bart Jacobs, and Frank Piessens

Katholieke Universiteit Leuven, Leuven, Belgium

{frederic.vogels,bart.jacobs,frank.piessens}@cs.kuleuven.be

**Abstract.** Machine-checked proofs of properties of programming languages have gained in importance significantly over the past few years. This paper contributes to this trend by proposing an approach for doing machine-checked soundness proofs for verification condition (VC) generators. Our approach embraces the multi-phase VC generation common in modern program verifiers. Such verifiers split the generation of VCs in two (or even more) phases, using an intermediate verification language as the bridge between the programming language and logic. In our approach, we define a formal operational semantics of the intermediate verification language, and we prove the soundness of two translations separately: (1) the translation of the intermediate verification language to VCs, and (2) the translation of the source programming language to the intermediate language. This paper presents a fully machine checked proof of step (1) for a prototypical intermediate verification language, and then illustrates step (2) for a very small object oriented programming language.

## 1 Introduction

Verification condition (VC) generation is one of the classic techniques for program verification: from the program and its specification one computes a set of logical sentences (the verification conditions) whose validity implies the correctness of the program with respect to the given specification. The technique can be traced back to the very roots of program verification [8].

However, as programming languages grew more and more complex with support for features such as dynamic memory allocation, pointers, exception handling mechanisms, objects, inheritance, dynamic binding and so forth, the process of generating VCs also became significantly more complex. To master this increased complexity, many modern program verifiers [5, 14] split the VC generation in two phases. First the source program and its specification are compiled to an intermediate verification language, and then VCs are generated from the intermediate language. A prominent example of such an intermediate language is the BoogiePL language [11, 4]. BoogiePL is the intermediate language of the Spec<sup>#</sup> program verifier [5] and the VCC verifying C compiler [12], and the ESC/Java line of verifiers is moving to a very similar intermediate language.

Another unfortunate consequence of the increased complexity of VC generation for current programming languages is that soundness proofs are either omitted (and hence the VC generation is seen as some form of axiomatic semantics of the source language [11]), or are presented only informally.

This paper proposes an approach to build modular machine-checked soundness proofs for VC generation. We embrace the two-phase VC generation implemented in modern verifiers, and we structure the soundness proof in the same way. Hence, in our approach, there are two translations that need to be proven sound: (1) the translation of the intermediate verification language to VCs, and (2) the translation of the source language to the intermediate language. This paper focuses on the first step, and gives a small example of the second step.

In Section 2, we give a short introduction to the  $\text{BoogiePL}^b$  intermediate verification language [11]. We then proceed to define its operational semantics and its VC generation algorithm. We conclude the first part with a proof of this algorithm's soundness with respect to the operational semantics. This proof is fully formalized and machine checked with the Coq proof assistant [6].

Section 3 contains a small example of step (2): we define a simple object oriented language with standard operational semantics and provide a translation to  $\text{BoogiePL}^b$ , which we also prove sound.

This technical report is an extended version of [13].

## 2 $\text{BoogiePL}^b$

The intermediate verification language we consider is essentially the same as the language defined by Leino and Schulte [11]. We only make slight modifications to the presentation of the syntax to make the formal proofs less cumbersome. The main novelty of this section (with respect to [11]) is the definition of the operational semantics of  $\text{BoogiePL}^b$  and the proof of the soundness of VC generation with respect to that operational semantics – in [11] the VC generation is considered an axiomatic definition of the semantics of the language.

In this section, we start off by introducing the reader to  $\text{BoogiePL}^b$  by means of an informal overview of the language (Section 2.1). Next, we formally define its syntax, operational semantics (Section 2.2) and VC generation algorithm (Section 2.3), which paves the way to a proof of  $\text{BoogiePL}^b$ 's soundness (Section 2.4).

### 2.1 Overview of the language

A  $\text{BoogiePL}^b$  program consists of two parts: on the one hand, a logical part that defines constants, function symbols and axioms. The constants and functions become part of the first-order logical signature over which the VCs are formulated. The signature plus the axioms constitute a classical logical theory. As this part is fairly standard, we refer the interested reader to [11] for a more detailed description. Suffice it to say that it has an axiomatization for integers, finite maps, booleans and so forth.

On the other hand there is the imperative part, that consists of (1) global variables which take values in the mathematical structure axiomatized by the logical part and represent (part of) the program state, and (2) a number of procedures that can be thought of as describing the possible control-flow paths in the program being verified. For example, as BoogiePL<sup>b</sup> does not provide a heap, a translation can instead define it as a global variable being a map of (object reference, field name) pairs to values.

Procedures are parameterized operations on the state space defined by the global variables. A procedure's body consists of a single command. The following commands are provided:

- sequential composition, written  $c_1; c_2$ .
- variable declaration, written **var**  $id : type$  which introduces a new variable with unknown initial value.
- **assert**  $expression$ : states that the expression must evaluate to true when execution passes that point, which can be used to specify proof obligations.
- **assume**  $expression$ : tells the verifier that the given expression can be assumed to be true, e.g. preconditions can be assumed to be true at the beginning of a procedure or postconditions to hold just after a procedure call.
- **havoc**  $identifier$ : the opposite of assume; it removes any information about the specified variable by assigning an arbitrary value to it.
- choice, written  $c_1 [] c_2$ : represents a control flow fork: execution could continue with either  $c_1$  or  $c_2$ . This command is typically used to model conditional branches, such as if- or while-statements.
- assignment, written  $x := expression$ : changes the variable  $x$ 's value.

Since procedure specifications are so common, BoogiePL<sup>b</sup> supports them directly: one can define procedure specifications followed by one or more procedure implementations, which all have to obey the specifications. The specifications consist of a number of **requires** clauses (the preconditions), **ensures** clauses (the postconditions), and a **modifies** clause, which indicates which global variables have their values changed by the procedure. Procedure definitions and calls can be treated as syntactic sugar, and hence we don't discuss them in this paper.

Details of how procedure calls can be desugared to the assertion of preconditions, a havoc on the modifies-variables followed by the assumption of postconditions can be found in [11]. We also omitted the block command which in [11] introduces a new variable scope. Blocks are not strictly necessary as it is possible to move all new local variable declaration to the method scope after having given them unique names.

## 2.2 Syntax and Operational semantics

We focus on the BoogiePL<sup>b</sup> commands, and leave expressions abstract. We only assume that expressions define side-effect free functions from the store to the set of values.

**Definition 1.** *The syntax is defined by the following production rules*

$$\begin{aligned}
\text{command} ::= & \mathbf{assert} \text{ expression} ; \text{command} \\
& | \mathbf{assume} \text{ expression} ; \text{command} \\
& | \mathbf{havoc} \text{ identifier} ; \text{command} \\
& | \text{identifier} := \text{expression} ; \text{command} \\
& | \text{command} [] \text{command} ; \text{command} \\
& | \mathbf{nil}
\end{aligned}$$

**Definition 2 (Appending).** We define the appending of commands, written  $c_1 \oplus c_2$ , as

$$\begin{aligned}
(\mathbf{assert} \ e; c) \oplus c' &= \mathbf{assert} \ e; (c \oplus c') \\
(\mathbf{assume} \ e; c) \oplus c' &= \mathbf{assume} \ e; (c \oplus c') \\
(\mathbf{havoc} \ x; c) \oplus c' &= \mathbf{havoc} \ x; (c \oplus c') \\
(x := e; c) \oplus c' &= x := e; (c \oplus c') \\
(c_1 [] c_2; c) \oplus c' &= c_1 [] c_2; (c \oplus c') \\
\mathbf{nil} \oplus c' &= c'
\end{aligned}$$

We define the operational semantics as a binary relation on machine states, written  $\sigma \rightsquigarrow \sigma'$ . We call this relation the single step relation as  $\sigma \rightsquigarrow \sigma'$  means that a machine in state  $\sigma$  can reach another state  $\sigma'$  in one step. Note the use of the verb “can”: in the context of nondeterministic semantics a state  $\sigma$  can lead to (perhaps infinitely) many other states. The machine can then switch over to any one of these states. This is the case with BoogiePL<sup>b</sup>, where some commands (such as **havoc** and  $[]$ ) behave nondeterministically.

We distinguish two kinds of state:

- A failure state (**fail**): this state indicates that the execution went wrong;
- An in-progress state, made out of two components, written  $(c \mid \mu)$ 
  - the command  $c$ : what’s left of the program to execute;
  - the store  $\mu$ : a total mapping from identifiers to values.

*Assert.* The **assert** command verifies whether a given expression evaluates to true. If it does, all is well and execution can proceed. Otherwise, the execution goes wrong and ends up in the failure state.

$$\frac{e[\mu] = \text{true}}{(\mathbf{assert} \ e; c \mid \mu) \rightsquigarrow (c \mid \mu)} \text{B-ASSERTTRUE}$$

$$\frac{e[\mu] \neq \text{true}}{(\mathbf{assert} \ e; c \mid \mu) \rightsquigarrow \mathbf{fail}} \text{B-ASSERTFALSE}$$

*Assume.* The **assume** command is used to “filter out” certain execution paths. If the assume condition evaluates to true in the current store, execution can proceed as if nothing happened. Otherwise, the current execution path needn’t be considered any further, which we achieve by not providing a rule for this case. Note that a state only fails if it leads to **fail**, hence an **assume** command with a false expression is not considered as failing.

$$\frac{e[\mu] = \text{true}}{(\mathbf{assume} \ e; c \mid \mu) \rightsquigarrow (c \mid \mu)} \text{ B-ASSUME}$$

*Havoc.* The **havoc** command destroys all information we have about a certain variable: **havoc**  $x$  indicates that the rest of the program must succeed for any value of  $x$ . This translates to having a nondeterministic reduction rule for this command: for every possible value  $v$ , we start a new execution path where  $x$  is bound to  $v$ , meaning there are infinitely many possible reductions for a state  $(\mathbf{havoc} \ x; c \mid \mu)$  because there are infinitely many values.

$$\overline{(\mathbf{havoc} \ x; c \mid \mu) \rightsquigarrow (c \mid \mu, x \mapsto v)} \text{ B-HAVOC}$$

*Assignment.* An assignment updates the store with a new binding.

$$\overline{(x := e; c \mid \mu) \rightsquigarrow (c \mid \mu, x \mapsto e[\mu])} \text{ B-ASSIGN}$$

*Choice.* This command represents a nondeterministic choice between two execution paths:  $c_1 \ [] \ c_2; c$  means that both  $c_1 \oplus c$  and  $c_2 \oplus c$  need to be considered.

$$\overline{(c_1 \ [] \ c_2; c \mid \mu) \rightsquigarrow (c_1 \oplus c \mid \mu)} \text{ B-CHOICELEFT}$$

$$\overline{(c_1 \ [] \ c_2; c \mid \mu) \rightsquigarrow (c_2 \oplus c \mid \mu)} \text{ B-CHOICERIGHT}$$

**Definition 3.** We define *BoogiePL<sup>b</sup>*'s operational semantics.

$$\frac{e[\mu] = \text{true}}{(\mathbf{assert} \ e; c \mid \mu) \rightsquigarrow (c \mid \mu)} \text{ B-ASSERTTRUE}$$

$$\frac{e[\mu] \neq \text{true}}{(\mathbf{assert} \ e; c \mid \mu) \rightsquigarrow \mathbf{fail}} \text{ B-ASSERTFALSE}$$

$$\frac{e[\mu] = \text{true}}{(\mathbf{assume} \ e; c \mid \mu) \rightsquigarrow (c \mid \mu)} \text{ B-ASSUME}$$

$$\overline{(\mathbf{havoc} \ x; c \mid \mu) \rightsquigarrow (c \mid \mu, x \mapsto v)} \text{ B-HAVOC}$$

$$\overline{(x := e; c \mid \mu) \rightsquigarrow (c \mid \mu, x \mapsto e[\mu])} \text{ B-ASSIGN}$$

$$\overline{(c_1 \ [] \ c_2; c \mid \mu) \rightsquigarrow (c_1 \oplus c \mid \mu)} \text{ B-CHOICELEFT}$$

$$\overline{(c_1 \ [] \ c_2; c \mid \mu) \rightsquigarrow (c_2 \oplus c \mid \mu)} \text{ B-CHOICERIGHT}$$

**Definition 4 (multiple step relation).**

$$\frac{}{\sigma \rightsquigarrow^* \sigma} \text{B}^*\text{-REFLEXIVE}$$

$$\frac{\sigma_1 \rightsquigarrow \sigma_2 \quad \sigma_2 \rightsquigarrow^* \sigma_3}{\sigma_1 \rightsquigarrow^* \sigma_3} \text{B}^*\text{-STEP}$$

**Definition 5.** We say a state  $\sigma$  fails if  $\sigma \rightsquigarrow^* \mathbf{fail}$ , i.e. there is an execution leading from  $\sigma$  to the failure state. In short, we write this as  $\text{fails } \sigma$ .

**Definition 6.** A state succeeds if it does not fail, written shortly as  $\text{succeeds } \sigma$ .

### 2.3 Weakest preconditions

**Definition 7 (Weakest preconditions).**

$$\begin{aligned} \text{wp}(\mathbf{assert } e; c, Q) &= e \wedge \text{wp}(c, Q) \\ \text{wp}(\mathbf{assume } e; c, Q) &= e \Rightarrow \text{wp}(c, Q) \\ \text{wp}(\mathbf{havoc } x; c, Q) &= \forall x \bullet \text{wp}(c, Q) \\ \text{wp}(x := e; c, Q) &= \text{wp}(c, Q)[e/x] \\ \text{wp}(c_1 [] c_2; c, Q) &= \text{wp}(c_1, \text{wp}(c, Q)) \wedge \text{wp}(c_2, \text{wp}(c, Q)) \\ \text{wp}(\mathbf{nil}, Q) &= Q \end{aligned}$$

**Lemma 1.**

$$\text{wp}(c_1 \oplus c_2, Q) = \text{wp}(c_1, \text{wp}(c_2, Q))$$

*Proof.* Structural induction on  $c_1$ :

–  $c_1 = \mathbf{assert } e; c$ . We need to show that

$$\text{wp}((\mathbf{assert } e; c) \oplus c_2, Q) = \text{wp}(\mathbf{assert } e; c, \text{wp}(c_2, Q)) \quad (1)$$

The left hand side of (1) becomes according to Definition 2

$$lhs = \text{wp}((\mathbf{assert } e; c) \oplus c_2, Q) = \text{wp}(\mathbf{assert } e; (c \oplus c_2), Q)$$

which is equal to (Definition 7)

$$lhs = e \wedge \text{wp}(c \oplus c_2, Q) \quad (2)$$

The right hand side of (1) becomes (Definition 7)

$$rhs = \text{wp}(\mathbf{assert } e; c, \text{wp}(c_2, Q)) = e \wedge \text{wp}(c, \text{wp}(c_2, Q)) \quad (3)$$

From the induction hypothesis:

$$\forall c_2, Q \bullet \text{wp}(c \oplus c_2, Q) = \text{wp}(c, \text{wp}(c_2, Q)) \quad (4)$$

Combining (3) and (4) gives

$$rhs = e \wedge \text{wp}(c, \text{wp}(c_2, Q)) = e \wedge \text{wp}(c \oplus c_2, Q) \quad (5)$$

The left hand side (2) and right hand side (5) are clearly equal.

–  $c_1 = \mathbf{assume} \ e; c$ . We need to show that

$$\text{wp}((\mathbf{assume} \ e; c) \oplus c_2, Q) = \text{wp}(\mathbf{assume} \ e; c, \text{wp}(c_2, Q)) \quad (6)$$

The left hand side of (6) becomes according to Definition 2

$$lhs = \text{wp}((\mathbf{assume} \ e; c) \oplus c_2, Q) = \text{wp}(\mathbf{assume} \ e; (c \oplus c_2), Q)$$

which is equal to (Definition 7)

$$lhs = e \Rightarrow \text{wp}(c \oplus c_2, Q) \quad (7)$$

The right hand side of (6) becomes (Definition 7)

$$rhs = \text{wp}(\mathbf{assume} \ e; c, \text{wp}(c_2, Q)) = e \Rightarrow \text{wp}(c, \text{wp}(c_2, Q)) \quad (8)$$

From the induction hypothesis:

$$\forall c_2, Q \bullet \text{wp}(c \oplus c_2, Q) = \text{wp}(c, \text{wp}(c_2, Q)) \quad (9)$$

Combining (8) and (9) gives

$$rhs = e \Rightarrow \text{wp}(c, \text{wp}(c_2, Q)) = e \Rightarrow \text{wp}(c \oplus c_2, Q) \quad (10)$$

The left hand side (7) and right hand side (10) are clearly equal.

–  $c_1 = \mathbf{havoc} \ x; c$ . We need to show that

$$\text{wp}((\mathbf{havoc} \ x; c) \oplus c_2, Q) = \text{wp}(\mathbf{havoc} \ x; c, \text{wp}(c_2, Q)) \quad (11)$$

The left hand side of (11) becomes according to Definition 2

$$lhs = \text{wp}((\mathbf{havoc} \ x; c) \oplus c_2, Q) = \text{wp}(\mathbf{havoc} \ x; (c \oplus c_2), Q)$$

which is equal to (Definition 7)

$$lhs = \forall x \bullet \text{wp}(c \oplus c_2, Q) \quad (12)$$

The right hand side of (11) becomes (Definition 7)

$$rhs = \text{wp}(\mathbf{havoc} \ x; c, \text{wp}(c_2, Q)) = \forall x \bullet \text{wp}(c, \text{wp}(c_2, Q)) \quad (13)$$

From the induction hypothesis:

$$\forall c_2, Q \bullet \text{wp}(c \oplus c_2, Q) = \text{wp}(c, \text{wp}(c_2, Q)) \quad (14)$$

Combining (13) and (14) gives

$$rhs = \forall x \bullet \text{wp}(c, \text{wp}(c_2, Q)) = \forall x \bullet \text{wp}(c \oplus c_2, Q) \quad (15)$$

The left hand side (12) and right hand side (15) are clearly equal.

–  $c_1 = x := e; c$ . We need to show that

$$\text{wp}((x := e; c) \oplus c_2, Q) = \text{wp}(x := e; c, \text{wp}(c_2, Q)) \quad (16)$$

The left hand side of (16) becomes according to Definition 2

$$\text{lhs} = \text{wp}((x := e; c) \oplus c_2, Q) = \text{wp}(x := e; (c \oplus c_2), Q)$$

which is equal to (Definition 7)

$$\text{lhs} = \text{wp}(c \oplus c_2, Q)[e/x] \quad (17)$$

The right hand side of (16) becomes (Definition 7)

$$\text{rhs} = \text{wp}(x := e; c, \text{wp}(c_2, Q)) = \text{wp}(c, \text{wp}(c_2, Q))[e/x] \quad (18)$$

From the induction hypothesis:

$$\forall c_2, Q \bullet \text{wp}(c \oplus c_2, Q) = \text{wp}(c, \text{wp}(c_2, Q)) \quad (19)$$

Combining (18) and (19) gives

$$\text{rhs} = \text{wp}(c, \text{wp}(c_2, Q)) = \text{wp}(c \oplus c_2, Q)[e/x] \quad (20)$$

The left hand side (17) and right hand side (20) are clearly equal.

–  $c_1 = c_a [] c_b; c$ . We need to show that

$$\text{wp}((c_a [] c_b; c) \oplus c_2, Q) = \text{wp}(c_a [] c_b; c, \text{wp}(c_2, Q)) \quad (21)$$

The left hand side of (21) becomes according to Definition 2

$$\text{lhs} = \text{wp}((c_a [] c_b; c) \oplus c_2, Q) = \text{wp}(c_a [] c_b; (c \oplus c_2), Q)$$

which is equal to (Definition 7)

$$\text{lhs} = \text{wp}(c_a, \text{wp}(c \oplus c_2, Q)) \wedge \text{wp}(c_b, \text{wp}(c \oplus c_2, Q)) \quad (22)$$

The right hand side of (21) becomes (Definition 7)

$$\begin{aligned} \text{rhs} &= \text{wp}(c_a [] c_b; c, \text{wp}(c_2, Q)) \\ &= \text{wp}(c_a, \text{wp}(c, \text{wp}(c_2, Q))) \wedge \text{wp}(c_b, \text{wp}(c, \text{wp}(c_2, Q))) \end{aligned} \quad (23)$$

From the induction hypothesis:

$$\forall c_2, Q \bullet \text{wp}(c \oplus c_2, Q) = \text{wp}(c, \text{wp}(c_2, Q)) \quad (24)$$

Combining (23) and (24) gives

$$\begin{aligned} \text{rhs} &= \text{wp}(c_a, \text{wp}(c, \text{wp}(c_2, Q))) \wedge \text{wp}(c_b, \text{wp}(c, \text{wp}(c_2, Q))) \\ &= \text{wp}(c_a, c \oplus c_2, Q) \wedge \text{wp}(c_b, c \oplus c_2, Q) \end{aligned} \quad (25)$$

The left hand side (22) and right hand side (25) are clearly equal.

–  $c_1 = \mathbf{nil}$ . We need to show that

$$\text{wp}(\mathbf{nil} \oplus c_2, Q) = \text{wp}(\mathbf{nil}, \text{wp}(c_2, Q)) \quad (26)$$

The left hand side becomes (Definition 2)

$$lhs = \text{wp}(\mathbf{nil} \oplus c_2, Q) = \text{wp}(c_2, Q) \quad (27)$$

while the right hand side becomes (Definition 7)

$$rhs = \text{wp}(\mathbf{nil}, \text{wp}(c_2, Q)) = \text{wp}(c_2, Q) \quad (28)$$

The left hand side (27) and right hand side (28) are clearly equal.  $\square$

## 2.4 Soundness of VC generation

**Definition 8 (Metric on commands).** We define the size of a command as follows

$$\begin{aligned} |\mathbf{assert} \ e; c| &= 1 + |c| \\ |\mathbf{assume} \ e; c| &= 1 + |c| \\ |\mathbf{havoc} \ x; c| &= 1 + |c| \\ |x := e; c| &= 1 + |c| \\ |c_1 \ [] \ c_2; c| &= 1 + |c_1| + |c_2| + |c| \\ |\mathbf{nil}| &= 0 \end{aligned}$$

**Lemma 2.** For all  $c_1, c_2$

$$|c_1 \oplus c_2| = |c_1| + |c_2|$$

*Proof.* Structural induction on  $c_1$ :

–  $c_1 = \mathbf{assert} \ e; c'$ . We need to show that

$$|(\mathbf{assert} \ e; c') \oplus c_2| = |\mathbf{assert} \ e; c'| + |c_2| \quad (29)$$

The left hand side becomes (Definition 2)

$$lhs = |(\mathbf{assert} \ e; c') \oplus c_2| = |\mathbf{assert} \ e; (c' \oplus c_2)|$$

which according to Definition 8 is equal to

$$lhs = 1 + |c' \oplus c_2| \quad (30)$$

The induction hypothesis tells us that

$$|c' \oplus c_2| = |c'| + |c_2|$$

Substituting in (30) leads to

$$lhs = 1 + |c'| + |c_2| \quad (31)$$

Definition 8 tells us the right hand side of (29) is equal to

$$rhs = |\mathbf{assert} \ e; c'| + |c_2| = 1 + |c'| + |c_2| \quad (32)$$

The left hand side (31) and right hand side (32) are clearly equal.

–  $c_1 = \mathbf{assume} \ e; c'$ . We need to show that

$$|(\mathbf{assume} \ e; c') \oplus c_2| = |\mathbf{assume} \ e; c'| + |c_2| \quad (33)$$

The left hand side becomes (Definition 2)

$$lhs = |(\mathbf{assume} \ e; c') \oplus c_2| = |\mathbf{assume} \ e; (c' \oplus c_2)|$$

which according to Definition 8 is equal to

$$lhs = 1 + |c' \oplus c_2| \quad (34)$$

The induction hypothesis tells us that

$$|c' \oplus c_2| = |c'| + |c_2|$$

Substituting in (34) leads to

$$lhs = 1 + |c'| + |c_2| \quad (35)$$

Definition 8 tells us the right hand side of (33) is equal to

$$rhs = |\mathbf{assume} \ e; c'| + |c_2| = 1 + |c'| + |c_2| \quad (36)$$

The left hand side (35) and right hand side (36) are clearly equal.

–  $c_1 = \mathbf{havoc} \ x; c'$ . We need to show that

$$|(\mathbf{havoc} \ x; c') \oplus c_2| = |\mathbf{havoc} \ x; c'| + |c_2| \quad (37)$$

The left hand side becomes (Definition 2)

$$lhs = |(\mathbf{havoc} \ x; c') \oplus c_2| = |\mathbf{havoc} \ x; (c' \oplus c_2)|$$

which according to Definition 8 is equal to

$$lhs = 1 + |c' \oplus c_2| \quad (38)$$

The induction hypothesis tells us that

$$|c' \oplus c_2| = |c'| + |c_2|$$

Substituting in (38) leads to

$$lhs = 1 + |c'| + |c_2| \quad (39)$$

Definition 8 tells us the right hand side of (37) is equal to

$$rhs = |\mathbf{havoc} \ e; c'| + |c_2| = 1 + |c'| + |c_2| \quad (40)$$

The left hand side (39) and right hand side (40) are clearly equal.

–  $c_1 = x := e; c'$ . We need to show that

$$|(x := e; c') \oplus c_2| = |x := e; c'| + |c_2| \quad (41)$$

The left hand side becomes (Definition 2)

$$lhs = |(x := e; c') \oplus c_2| = |x := e; (c' \oplus c_2)|$$

which according to Definition 8 is equal to

$$lhs = 1 + |c' \oplus c_2| \quad (42)$$

The induction hypothesis tells us that

$$|c' \oplus c_2| = |c'| + |c_2|$$

Substituting in (42) leads to

$$lhs = 1 + |c'| + |c_2| \quad (43)$$

Definition 8 tells us the right hand side of (41) is equal to

$$rhs = |x := e; c'| + |c_2| = 1 + |c'| + |c_2| \quad (44)$$

The left hand side (43) and right hand side (44) are clearly equal.

–  $c_1 = (c_a [] c_b); c'$ . We need to show that

$$|(c_a [] c_b; c') \oplus c_2| = |c_a [] c_b; c'| + |c_2| \quad (45)$$

The left hand side becomes (Definition 2)

$$lhs = |(c_a [] c_b; c') \oplus c_2| = |c_a [] c_b; (c' \oplus c_2)|$$

which according to Definition 8 is equal to

$$lhs = 1 + |c_a| + |c_b| + |c' \oplus c_2| \quad (46)$$

The induction hypothesis tells us that

$$|c' \oplus c_2| = |c'| + |c_2|$$

Substituting in (46) leads to

$$lhs = 1 + |c_a| + |c_b| + |c'| + |c_2| \quad (47)$$

Definition 8 tells us the right hand side of (45) is equal to

$$rhs = |c_a [] c_b; c'| + |c_2| = 1 + |c_a| + |c_b| + |c'| + |c_2| \quad (48)$$

The left hand side (47) and right hand side (48) are clearly equal.

–  $c_1 = \mathbf{nil}$ . We need to show that

$$|\mathbf{nil} \oplus c_2| = |\mathbf{nil}| + |c_2| \quad (49)$$

The left hand side equals (Definition 2)

$$lhs = |c_2| \quad (50)$$

The right hand side of (49) becomes

$$rhs = 0 + |c_2| = |c_2| \quad (51)$$

Both sides (50) and (51) are clearly equal.  $\square$

**Lemma 3.** *For each  $c_1, c_2, c$*

$$\begin{aligned} |c_1 \oplus c| &< |c_1 [] c_2; c| \\ |c_2 \oplus c| &< |c_1 [] c_2; c| \end{aligned}$$

*Proof.* We deal with both cases at the same time. We need to show that

$$|c_{1,2} \oplus c| < |c_1 [] c_2; c|$$

Applying Lemma 2 leads to a new goal

$$|c_{1,2}| + |c| < |c_1 [] c_2; c|$$

Definition 8 gives

$$|c_{1,2}| + |c| < 1 + |c_1| + |c_2| + |c|$$

which simplifies to

$$|c_{1,2}| < 1 + |c_1| + |c_2|$$

which is clearly true for both  $c_1$  and  $c_2$ , as command sizes are always positive.  $\square$

**Lemma 4 (Induction scheme on commands).** *For any predicate  $P$  over commands,*

$$\begin{aligned} (\forall c, e \bullet P(c) \Rightarrow P(\mathbf{assert} e; c)) &\Rightarrow \\ (\forall c, e \bullet P(c) \Rightarrow P(\mathbf{assume} e; c)) &\Rightarrow \\ (\forall c, x \bullet P(c) \Rightarrow P(\mathbf{havoc} x; c)) &\Rightarrow \\ (\forall c, x, e \bullet P(c) \Rightarrow P(x := e; c)) &\Rightarrow \\ (\forall c, c_1, c_2 \bullet P(c_1 \oplus c) \Rightarrow P(c_2 \oplus c) \Rightarrow P(c_1 [] c_2; c)) &\Rightarrow \\ P(\mathbf{nil}) \Rightarrow \forall c \bullet P(c) & \end{aligned}$$

*Proof.* We use induction on size

$$(\forall c \bullet (\forall c' \bullet |c'| < |c| \Rightarrow P(c')) \Rightarrow P(c)) \Rightarrow \forall c \bullet P(c) \quad (52)$$

meaning we have to prove that

$$\forall c \bullet (\forall c' \bullet |c'| < |c| \Rightarrow P(c')) \Rightarrow P(c) \quad (53)$$

given

$$\forall c, e \bullet P(c) \Rightarrow P(\mathbf{assert} \ e; c) \quad (54)$$

$$\forall c, e \bullet P(c) \Rightarrow P(\mathbf{assume} \ e; c) \quad (55)$$

$$\forall c, x \bullet P(c) \Rightarrow P(\mathbf{havoc} \ x; c) \quad (56)$$

$$\forall c, x, e \bullet P(c) \Rightarrow P(x := e; c) \quad (57)$$

$$\forall c, c_1, c_2 \bullet P(c_1 \oplus c) \Rightarrow P(c_2 \oplus c) \Rightarrow P(c_1 [] c_2; c) \quad (58)$$

$$P(\mathbf{nil}) \quad (59)$$

We distinguish the following cases for  $c$ :

- $c = \mathbf{assert} \ e; c'$ . We need to show that

$$P(\mathbf{assert} \ e; c')$$

According to Definition 8

$$|c| = |\mathbf{assert} \ e; c'| = 1 + |c'| > |c'|$$

hence, (53) leads to

$$P(c')$$

(54) gives

$$P(\mathbf{assert} \ e; c')$$

which is what we needed to prove.

- $c = \mathbf{assume} \ e; c'$ . We need to show that

$$P(\mathbf{assume} \ e; c')$$

According to Definition 8

$$|c| = |\mathbf{assume} \ e; c'| = 1 + |c'| > |c'|$$

hence, (53) leads to

$$P(c')$$

(55) gives

$$P(\mathbf{assume} \ e; c')$$

which is what we needed to prove.

- $c = \mathbf{havoc} \ x; c'$ . We need to show that

$$P(\mathbf{havoc} \ x; c')$$

According to Definition 8

$$|c| = |\mathbf{havoc} \ x; c'| = 1 + |c'| > |c'|$$

hence, (53) leads to

$$P(c')$$

(56) gives

$$P(\mathbf{havoc} \ x; c')$$

which is what we needed to prove.

–  $c = x := e; c'$ . We need to show that

$$P(x := e; c')$$

According to Definition 8

$$|c| = |x := e; c'| = 1 + |c'| > |c'|$$

hence, (53) leads to

$$P(c')$$

(57) gives

$$P(x := e; c')$$

which is what we needed to prove.

–  $c = c_1 [] c_2; c'$ . We need to show that

$$P(c_1 [] c_2; c')$$

We know from Lemma 3

$$\begin{aligned} |c_1 \oplus c'| &< |c_1 [] c_2; c'| = |c| \\ |c_2 \oplus c'| &< |c_1 [] c_2; c'| = |c| \end{aligned}$$

(53) gives then

$$\begin{aligned} P(c_1 \oplus c') \\ P(c_2 \oplus c') \end{aligned}$$

and (58) leads to

$$P(c_1 [] c_2; c')$$

–  $c = \mathbf{nil}$ . The goal

$$P(\mathbf{nil})$$

follows directly from (59). □

**Theorem 1 (Soundness).** *Given a program  $c$ , its evaluation does not fail if the initial state  $\mu$  fulfills the weakest precondition.*

$$\forall c, \mu \bullet \text{wp}(c, \text{true})[\mu] \Rightarrow \text{succeeds}(c \mid \mu)$$

*Proof.* We make use of the induction scheme defined in Lemma 4 where we take

$$P(c) = \forall \mu \bullet \text{wp}(c, \text{true})[\mu] \Rightarrow \text{succeeds}(c \mid \mu)$$

We need to prove the following:

- For all  $c, e$ , given (see Lemma 4)

$$\forall \mu \bullet \text{wp}(c, \text{true})[\mu] \Rightarrow \text{succeeds}(c \mid \mu) \quad (60)$$

and for all  $\mu$

$$\text{wp}(\mathbf{assert} \ e; c, \text{true})[\mu] \quad (61)$$

we must show that

$$\text{succeeds}(\mathbf{assert} \ e; c \mid \mu)$$

From (61) and Definition 7 we know that

$$\text{wp}(\mathbf{assert} \ e; c, \text{true})[\mu] = (e \wedge \text{wp}(c, \text{true}))[\mu]$$

from which follows

$$e[\mu] \quad (62)$$

$$\text{wp}(c, \text{true})[\mu] \quad (63)$$

From (60) and (63) follows

$$\text{succeeds}(c \mid \mu) \quad (64)$$

Considering the state  $(\mathbf{assert} \ e; c \mid \mu)$ , only B-ASSERTTRUE applies as (62) prohibits B-ASSERTFALSE from happening:

$$(\mathbf{assert} \ e; c \mid \mu) \rightsquigarrow (c \mid \mu)$$

From (64) we know that  $(c \mid \mu)$  will not fail, so neither will  $(\mathbf{assert} \ e; c \mid \mu)$ , which is what we needed to prove.

- For all  $c, e$ , given (see Lemma 4)

$$\forall \mu \bullet \text{wp}(c, \text{true})[\mu] \Rightarrow \text{succeeds}(c \mid \mu) \quad (65)$$

and for all  $\mu$

$$\text{wp}(\mathbf{assume} \ e; c, \text{true})[\mu] \quad (66)$$

we must show that

$$\text{succeeds}(\mathbf{assume} \ e; c \mid \mu)$$

From (61) and Definition 7 we know that

$$\text{wp}(\mathbf{assume} \ e; c, \text{true})[\mu] = (e \Rightarrow \text{wp}(c, \text{true}))[\mu]$$

from which follows

$$e[\mu] \Rightarrow \text{wp}(c, \text{true})[\mu] \quad (67)$$

$e[\mu]$  is either true or false, we consider both possibilities separately:

- $e[\mu]$ , in which case B-ASSUME applies on  $(\mathbf{assume} \ e; c \mid \mu)$ :

$$(\mathbf{assume} \ e; c \mid \mu) \rightsquigarrow (c \mid \mu)$$

We also know from (67) that

$$\text{wp}(c, \text{true})[\mu]$$

from which follows because of (65)

$$\text{succeeds}(c \mid \mu)$$

meaning  $(\mathbf{assume} \ e; c \mid \mu)$  will not fail through this path.

- $\neg e[\mu]$ , in which case  $(\mathbf{assume} \ e; c \mid \mu)$  is stuck and thus doesn't fail.

By considering every execution path beginning from  $(\mathbf{assume} \ e; c \mid \mu)$  and showing none of them fails, we are done with this part of the proof.

- For all  $c, e$ , given (see Lemma 4)

$$\forall \mu \bullet \text{wp}(c, \text{true})[\mu] \Rightarrow \text{succeeds}(c \mid \mu) \quad (68)$$

and for all  $\mu$

$$\text{wp}(\mathbf{havoc} \ x; c, \text{true})[\mu] \quad (69)$$

we must show that

$$\text{succeeds}(\mathbf{havoc} \ x; c \mid \mu)$$

From (69) and Definition 7 we know that

$$\text{wp}(\mathbf{havoc} \ x; c, \text{true})[\mu] = (\forall x \bullet \text{wp}(c, \text{true}))[\mu] \quad (70)$$

We know from Definition 3 that, for all  $v$

$$(\mathbf{havoc} \ x; c \mid \mu) \rightsquigarrow (c \mid \mu, x \mapsto v)$$

We wish to show that the following is true

$$\text{wp}(c, \text{true})[\mu, x \mapsto v]$$

which we can also write as

$$\text{wp}(c, \text{true})[v/x][\mu]$$

We know from (70) that this proposition is indeed true. Thus, according to (68)

$$\text{succeeds}(c \mid \mu, x \mapsto v)$$

meaning that every state reachable from our initial state  $(\mathbf{havoc} \ x; c \mid \mu)$  succeeds.

– For all  $c, e$ , given (see Lemma 4)

$$\forall \mu \bullet \text{wp}(c, \text{true})[\mu] \Rightarrow \text{succeeds}(c \mid \mu) \quad (71)$$

and for all  $\mu$

$$\text{wp}(x := e; c, \text{true})[\mu] \quad (72)$$

we must show that

$$\text{succeeds}(x := e; c \mid \mu)$$

The only applicable rule on  $(x := e; c \mid \mu)$  is B-SET:

$$(x := e; c \mid \mu) \rightsquigarrow (c \mid \mu, x \mapsto e(\mu))$$

We now need to show that

$$\text{succeeds}(c \mid \mu, x \mapsto e(\mu))$$

According to (71), this is the case if

$$\text{wp}(c, \text{true})[\mu, x \mapsto e(\mu)]$$

which is equal to

$$\text{wp}(c, \text{true})[e(\mu)/x][\mu] \quad (73)$$

By Definition 7, (72) is equal to

$$\text{wp}(x := e; c, \text{true})[\mu] = \text{wp}(c, \text{true})[e/x][\mu]$$

which is equal to (73), which is what we needed to show.

– For all  $c, e$ , given (see Lemma 4)

$$\forall \mu \bullet \text{wp}(c_1 \oplus c, \text{true})[\mu] \Rightarrow \text{succeeds}(c_1 \oplus c \mid \mu) \quad (74)$$

$$\forall \mu \bullet \text{wp}(c_2 \oplus c, \text{true})[\mu] \Rightarrow \text{succeeds}(c_2 \oplus c \mid \mu) \quad (75)$$

and for all  $\mu$

$$\text{wp}(c_1 [] c_2; c, \text{true})[\mu] \quad (76)$$

we must show that

$$\text{succeeds}(c_1 [] c_2; c \mid \mu)$$

Starting from  $(c_1 [] c_2; c \mid \mu)$ , only two reduction rules apply:

- B-CHOICELEFT

$$(c_1 [] c_2; c \mid \mu) \rightsquigarrow (c_1 \oplus c \mid \mu)$$

- B-CHOICERIGHT

$$(c_1 [] c_2; c \mid \mu) \rightsquigarrow (c_2 \oplus c \mid \mu)$$

We thus need to prove that both

$$\text{succeeds } (c_1 \oplus c \mid \mu) \text{succeeds } (c_2 \oplus c \mid \mu)$$

For this, we need to show (using (74) and (75))

$$\begin{aligned} & \text{wp}(c_1 \oplus c, \text{true})[\mu] \\ & \text{wp}(c_2 \oplus c, \text{true})[\mu] \end{aligned}$$

According to Definition 7, (72) becomes

$$\text{wp}(c_1 [] c_2; c, \text{true})[\mu] = (\text{wp}(c_1, \text{wp}(c, \text{true})) \wedge \text{wp}(c_2, \text{wp}(c, \text{true})))[\mu]$$

which becomes

$$\begin{aligned} & \text{wp}(c_1, \text{wp}(c, \text{true}))[\mu] \\ & \text{wp}(c_2, \text{wp}(c, \text{true}))[\mu] \end{aligned}$$

Applying Lemma 1 leads to

$$\begin{aligned} & \text{wp}(c_1 \oplus c, \text{true}) \\ & \text{wp}(c_2 \oplus c, \text{true}) \end{aligned}$$

which is what we needed to prove.

– We must show that

$$\text{succeeds } (\text{nil} \mid \mu)$$

This is trivially true, as no reduction rule applies, meaning it is a stuck state, which by Definition 6 succeeds.

□

### 3 Translating to BoogiePL<sup>b</sup>

Building a verifier for a programming language can be done by compiling the language to BoogiePL<sup>b</sup> and then verifying the resulting BoogiePL<sup>b</sup> program. For the resulting verifier to be sound, the compilation of the programming language to BoogiePL<sup>b</sup> needs to be correct. In this section, we describe the compilation and the correctness proof for a very small object-oriented programming language. It is a very simple class-based language with pre- and postcondition declarations as specification constructs.

#### 3.1 A toy OO language with contracts

The source programming language that we start from contains both programming constructs as well as specification constructs. The syntax of the language is summarized in Figure 1.

The semantics of the language is defined using a straightforward small-step operational semantics. The operational semantics defines all the erroneous program behaviour that the verifier is supposed to catch as going to a special

```

Program ::= Class*
Class ::= class Identifier { Field* Method* }
Field ::= field Identifier : Type
Method ::= method Identifier ( Argument* ) : Type Spec { Stm* }
Argument ::= Identifier : Type
Spec ::= requires Precondition ensures Postcondition
Stm ::= local Identifier : Type ;
      | Identifier = null ;
      | Identifier = Identifier . Identifier ;
      | Identifier = Identifier . Type :: Identifier ( Identifier* ) ;
      | Identifier = new Identifier ;
      | Identifier.Identifier = Identifier ;
      | if ( Identifier == Identifier ) Stm else Stm
Type ::= Identifier

```

**Fig. 1.** Syntax of the toy OO language

FAIL state. Other program errors that are caught by other means – for instance method-not-understood errors caught by the type system – can cause the operational semantics to get stuck. For the purpose of this paper, we don’t care about these errors: our soundness theorem will just say that a program that verifies will never go to the FAIL state.

The program state is represented as a quintuplet:

- The heap: a partial function mapping object identifiers (*oids*) to objects, which themselves are partial functions mapping field names (identifiers) to *oids*.

$$H : oid \rightarrow fieldname \rightarrow oid$$

- The store: a stack of frames, which are partial functions mapping identifiers to *oids*. It is used to store local variable bindings.
- The postcondition stack keeps track of method postconditions which need to be checked at method exit.
- The receiver stack keeps track of identifiers denoting local variables which receive the return value on method exit.
- The call stack containing statement lists; it acts as the program counter.

Pre- and postconditions are classical logical propositions for which we do not define a syntax; instead we choose to abstract them away as functions. A method  $m$ ’s precondition  $Pre_m$  needs access to its arguments and objects on the heap, which is why we model preconditions as binary functions receiving the upper frame (which contains the arguments) and the heap. The postconditions ( $Post_m$ ) can refer to the arguments, the heap state before and after the method call, and the return value. Therefore we model postconditions as functions taking an upper frame, a before-heap, an after-heap, and a return value. We’ll also use partial application on postconditions; an example of this can be found in E-METHODCALL, where we only apply  $H$  and  $F$ , while the rest of the arguments will have to wait until E-EXITMETHOD.

**Definition 9 (Initial state).** *Given a program  $P$  where some class  $X$  is the only to contain a nullary preconditionless  $T$ -returning method named `main`, the initial state  $\sigma_{\text{init}}(P)$  is equal to*

$$(\epsilon, \epsilon \circ \epsilon, \epsilon, \epsilon, \bar{s} \circ \epsilon)$$

where

$$\begin{aligned} \bar{s} &= \mathbf{local} \ x:X; \\ &\quad x = \mathbf{new} \ X; \\ &\quad \mathbf{local} \ r : T; \\ &\quad r = x.X::\mathbf{main}(); \end{aligned}$$

A verifier is supposed to check that it is not the case that  $\sigma_{\text{init}}(P) \longrightarrow^* \mathbf{fail}$ . We design a verifier by translation to BoogiePL<sup>b</sup> and we give a soundness proof of that translation.

### 3.2 Reduction rules

We first thoroughly discuss each reduction rule in turn, after which we summarize them in a table.

- E-LOCAL: introduces a new local variable which is initialized to **null**.

$$\begin{aligned} (H, F \circ Fs, Cs, Rs, (\mathbf{local} \ x:T; \bar{s}) \circ Ps) \\ \downarrow \\ (H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

- E-STORENULL: writes **null** to a local variable.

$$(H, F \circ Fs, Cs, Rs, (x=\mathbf{null}; \bar{s}) \circ Ps) \longrightarrow (H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps)$$

- E-STOREFIELD: fetches a field value and stores it in a local variable.

$$\begin{aligned} (H, F \circ Fs, Cs, Rs, (x=y.f; \bar{s}) \circ Ps) \\ \downarrow \\ (H, F' \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

where

$$\begin{aligned} oid &= F[y] \neq \mathbf{null} \\ F' &= F, x \mapsto H[oid][f] \end{aligned}$$

- E-STOREFIELDNULL: reading a field through a **null** reference results in failure.

$$(H, F \circ Fs, Cs, Rs, (x=y.f; \bar{s}) \circ Ps) \longrightarrow \mathbf{FAIL}$$

where

$$F[y] = \mathbf{null}$$

- E-STORENEW: creates a new object and stores a reference to it in a local variable.

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x=\mathbf{new} K; \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H', (F', x \mapsto oid) \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

where

$$\begin{aligned} & oid \notin H \\ & H' = H, oid \mapsto \mathbb{F} \\ & \mathbb{F} = \epsilon, \bar{f} \mapsto \mathbf{null} \end{aligned}$$

- E-WRITEFIELD: writes a value to an object field.

$$(H, F \circ Fs, Cs, Rs, (x.f = y; \bar{s}) \circ Ps) \longrightarrow (H', F \circ Fs, Cs, Rs, \bar{s} \circ Ps)$$

where

$$\begin{aligned} & oid = F[y] \neq \mathbf{null} \\ & \mathbb{F} = H[oid] \\ & v = F[y] \\ & \mathbb{F}' = \mathbb{F}, f \mapsto v \\ & H' = H, oid \mapsto \mathbb{F}' \end{aligned}$$

- E-WRITEFIELDNULL: attempts to write to a field through a **null** reference results in failure.

$$(H, F \circ Fs, Cs, Rs, x.f = y; \bar{s} \circ Ps) \longrightarrow \mathbf{FAIL}$$

where

$$F[x] = \mathbf{null}$$

- E-IFTRUE: if the condition is true, the **then**-branch must be executed next.

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (\mathbf{if} (x == y) s_1 \mathbf{else} s_2 \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F \circ Fs, Cs, Rs, (s_1 \bar{s}) \circ Ps) \end{aligned}$$

where

$$F[x] = F[y]$$

- E-IFFALSE: if the condition is false, the **else** branch must be executed next.

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (\mathbf{if} (x == y) s_1 \mathbf{else} s_2 \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F \circ Fs, Cs, Rs, (s_2 \bar{s}) \circ Ps) \end{aligned}$$

where

$$F[x] \neq F[y]$$

- E-METHODCALL: method invocation. The new body is added to the state stack, a new stack frame is created containing bindings for the **this** reference, method arguments and **result** variable. The method’s postcondition is pushed on the condition stack with the before-state already filled in. The receiver variable (the one to which the method result will be written to upon return) is pushed on the receiver stack.

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x=y.T::m(\bar{z}) \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F' \circ F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs, body_{T::m} \circ \bar{s} \circ Ps) \end{aligned}$$

where

$$\begin{aligned} F = \epsilon, \mathbf{this} &\mapsto oid, \overline{args_{T::m}} \mapsto F[\bar{z}], \mathbf{result} \mapsto \mathbf{null} \\ oid &= F[y] \neq \mathbf{null} \\ Pre_{T::m}(H, F) &= true \end{aligned}$$

- E-METHODCALLFAIL: calling a method whose preconditions aren’t satisfied yields failure.

$$(H, F \circ Fs, Cs, Rs, x=y.T::m(\bar{z}) \bar{s} \circ Ps) \longrightarrow \text{FAIL}$$

where

$$Pre_{T::m}(H, F) \neq true$$

- E-METHODCALLNULL: invoking a method through a **null** reference leads to failure.

$$(H, F \circ Fs, Cs, Rs, x=y.T::m(\bar{z}) \bar{s} \circ Ps) \longrightarrow \text{FAIL}$$

where

$$F[y] = \mathbf{null}$$

- E-EXITMETHOD: pops the necessary stacks and writes the **result** value to the receiver variable.

$$\begin{aligned} & (H, F' \circ F \circ Fs, C \circ Cs, x \circ Rs, \epsilon \circ Ps) \\ & \quad \downarrow \\ & (H, (F, x \mapsto F'[\mathbf{result}]) \circ Fs, Cs, Rs, Ps) \end{aligned}$$

where

$$C(H, F'[\mathbf{result}]) = true$$

- E-EXITMETHODFAIL: exiting a method fails if the method’s postcondition isn’t satisfied.

$$(H, F \circ Fs, C \circ Cs, Rs, \epsilon \circ \bar{s} \circ Ps) \longrightarrow \text{FAIL}$$

where

$$C(H, F[\mathbf{result}]) \neq true$$

**Definition 10 (Single step relation).**

|                  |  |   |
|------------------|--|---|
| E-LOCAL          | $(H, F \circ Fs, Cs, Rs, (\mathbf{local} \ x:\mathbf{T}; \bar{s}) \circ Ps)$<br>$\longrightarrow (H, F' \circ Fs, Cs, Rs, \bar{s} \circ Ps)$   | $F' = F, x \mapsto \mathbf{null}$   |
| E-STORENULL      | $(H, F \circ Fs, Cs, Rs, (x=\mathbf{null}; \bar{s}) \circ Ps)$<br>$\longrightarrow (H, F' \circ Fs, Cs, Rs, \bar{s} \circ Ps)$   | $F' = F, x \mapsto \mathbf{null}$   |
| E-STOREFIELD     | $(H, F \circ Fs, Cs, Rs, (x=y.f; \bar{s}) \circ Ps)$<br>$\longrightarrow (H, F' \circ Fs, Cs, Rs, \bar{s} \circ Ps)$   | $oid = F[y] \neq \mathbf{null}$<br>$F' = F, x \mapsto H[oid][f]$  |
| E-STOREFIELDNULL | $(H, F \circ Fs, Cs, Rs, (x=y.f; \bar{s}) \circ Ps) \longrightarrow \text{FAIL}$   | $F[y] = \mathbf{null}$  |
| E-STORENEW       | $(H, F \circ Fs, Cs, Rs, (x=\mathbf{new} \ K; \bar{s}) \circ Ps)$<br>$\longrightarrow (H', (F', x \mapsto oid) \circ Fs, Cs, Rs, \bar{s} \circ Ps)$  | $oid \notin H, H' = H, oid \mapsto \mathbb{F}$<br>$\mathbb{F} = \epsilon, \bar{f} \mapsto \mathbf{null}$  |
| E-WRITEFIELD     | $(H, F \circ Fs, Cs, Rs, (x.f = y; \bar{s}) \circ Ps)$<br>$\longrightarrow (H', F \circ Fs, Cs, Rs, \bar{s} \circ Ps)$   | $oid = F[y] \neq \mathbf{null}$<br>$\mathbb{F} = H[oid], v = F[y]$<br>$F' = \mathbb{F}, f \mapsto v$<br>$H' = H, oid \mapsto \mathbb{F}'$   |
| E-WRITEFIELDNULL | $(H, F \circ Fs, Cs, Rs, x.f = y; \bar{s} \circ Ps) \longrightarrow \text{FAIL}$   | $F[x] = \mathbf{null}$  |
| E-IFTRUE         | $(H, F \circ Fs, Cs, Rs, (\mathbf{if} \ (x == y) \ s_1 \ \mathbf{else} \ s_2 \ \bar{s}) \circ Ps)$<br>$\longrightarrow (H, F \circ Fs, Cs, Rs, (s_1 \ \bar{s}) \circ Ps)$                        | $F[x] = F[y]$   |
| E-IFFALSE        | $(H, F \circ Fs, Cs, Rs, (\mathbf{if} \ (x == y) \ s_1 \ \mathbf{else} \ s_2 \ \bar{s}) \circ Ps)$<br>$\longrightarrow (H, F \circ Fs, Cs, Rs, (s_2 \ \bar{s}) \circ Ps)$                        | $F[x] \neq F[y]$  |
| E-METHODCALL     | $(H, F \circ Fs, Cs, Rs, (x=y.T::m(\bar{z}) \ \bar{s}) \circ Ps) \longrightarrow$<br>$(H, F' \circ F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs,$<br>$body_{T::m} \circ \bar{s} \circ Ps)$ | $oid = F[y] \neq \mathbf{null}$<br>$Pre_{T::m}(H, F) = \mathbf{true}$<br>$F' = \epsilon, \mathbf{this} \mapsto oid,$<br>$\overline{args_{T::m}} \mapsto F[\bar{z}],$<br>$\mathbf{result} \mapsto \mathbf{null}$ |
| E-METHODCALLFAIL | $(H, F \circ Fs, Cs, Rs, x=y.T::m(\bar{z}) \ \bar{s} \circ Ps) \longrightarrow \text{FAIL}$  | $Pre_{T::m}(H, F) \neq \mathbf{true}$   |
| E-METHODCALLNULL | $(H, F \circ Fs, Cs, Rs, x=y.T::m(\bar{z}) \ \bar{s} \circ Ps) \longrightarrow \text{FAIL}$  | $F[y] = \mathbf{null}$  |
| E-EXITMETHOD     | $(H, F' \circ F \circ Fs, C \circ Cs, x \circ Rs, \epsilon \circ Ps)$<br>$\longrightarrow (H, (F, x \mapsto F'[\mathbf{result}]) \circ Fs, Cs, Rs, Ps)$  | $R = x \circ R'$<br>$C(H, F'[\mathbf{result}]) = \mathbf{true}$   |
| E-EXITMETHODFAIL | $(H, F \circ Fs, C \circ Cs, Rs, \epsilon \circ \bar{s} \circ Ps) \longrightarrow \text{FAIL}$   | $C(H, F[\mathbf{result}]) \neq \mathbf{true}$   |

**Definition 11 (Multiple step relation).** We define a binary multiple step relation as follows:

$$\frac{}{\sigma \longrightarrow^* \sigma} \text{E}^*\text{-REFLEXIVE}$$

$$\frac{\sigma_1 \longrightarrow \sigma_2 \quad \sigma_2 \longrightarrow^* \sigma_3}{\sigma_1 \longrightarrow^* \sigma_3} \text{E}^*\text{-STEP}$$

**Failure** The reduction rules show in which way a failure state can be reached. We summarize them here.

- E-STORENULL: reading a field through a **null** reference, for example

$x = \mathbf{null};$   
 $y = x.\mathbf{field};$

will inevitably lead to failure.

- E-STOREFIELDNULL: writing to a field through a **null** reference, for example
 
$$\begin{aligned} x &= \mathbf{null}; \\ x.f &= y; \end{aligned}$$
- E-METHODCALLNULL: calling a method on a null reference, for example
 
$$\begin{aligned} x &= \mathbf{null}; \\ y &= x.m(); \end{aligned}$$
- E-METHODCALLFAIL: calling a method when its preconditions are not satisfied.
- E-EXITMETHODFAIL: returning from a method when its postconditions are not satisfied.

These are the failures we wish to prevent using verification condition generation. If a program verifies, it means that none of these failures will be encountered at runtime.

**Theorem 2.** *The reduction rules (Section 3.2) are deterministic.*

$$\forall \sigma, \sigma_1, \sigma_2 \bullet \sigma \longrightarrow \sigma_1 \Rightarrow \sigma \longrightarrow \sigma_2 \Rightarrow \sigma_1 = \sigma_2$$

*Proof.* We consider an arbitrary state  $(H, Fs, Cs, Rs, \bar{s} \circ Ps)$ . The assumption that the fifth component of the quintuplet has at least one element is sound, as a state  $(H, Fs, Cs, Rs, \epsilon)$  is stuck (i.e. there are no applicable reduction rules). We consider the different forms  $\bar{s}$  can take:

- $\bar{s} = \mathbf{local } x:T; \bar{s}'$ : only E-LOCAL applies.
- $\bar{s} = x=\mathbf{null}; \bar{s}'$ : only E-STORENULL applies.
- $\bar{s} = x=y.f; \bar{s}'$ : a choice has to be made between E-STOREFIELD and E-STOREFIELDNULL; the former is only applicable when  $F[y] = \mathbf{null}$ , the latter when  $F[y] \neq \mathbf{null}$ . Since both can't be true at the same time, we have determinism.
- $\bar{s} = x=\mathbf{new}T; \bar{s}'$ : only E-STORENEW applies.
- $\bar{s} = x.f=y; \bar{s}'$ : the choice between E-WRITEFIELD and E-WRITEFIELDNULL depends on whether  $F[y]$  is equal to **null** or not.
- $\bar{s} = \mathbf{if } (x==y) s_1 \mathbf{ else } s_2 \bar{s}'$ : the choice between E-IFTRUE and E-IFFALSE depends on whether or not  $F[x] = F[y]$ .
- $\bar{s} = \epsilon$ : the choice between E-EXITMETHOD and E-EXITMETHODFAIL is made by determining whether a certain condition is true or false (i.e. whether that method's postcondition is satisfied or not).
- $\bar{s} = x=y.T::m(\bar{z}); \bar{s}'$ : the rules E-METHODCALL, E-METHODCALLFAIL and E-METHODCALLNULL deal with the method invocation statement. As both E-METHODCALLFAIL and E-METHODCALLNULL lead to FAIL, we don't need to find a discriminating condition for those. We can make a deterministic choice between E-METHODCALL and E-METHODCALLNULL based on whether  $F[y]$  is equal to **null** or not. The choice between E-METHODCALL and E-METHODCALLFAIL depends on a condition (i.e. the method's precondition) being true or not.

□

### 3.3 Modularization

As the size of verification conditions grows quickly with the size of the BoogiePL<sup>b</sup> program, it is essential for scalability that the verification can be done modularly. For the language under consideration, we will do verification per method: each method will be verified in isolation, using only the method body and the specifications of the methods that this method calls.

We introduce the notion of state lifting which we will use to show that only the top elements of the stacks in the machine state influence execution, i.e. the behaviour of a method does not depend on how deep it is in the call stack.

**Definition 12 (State lifting).**

$$\frac{(H, Fs, Cs, Rs, Ps)}{(Fs_+, Cs_+, Rs_+, Ps_+)} = (H, Fs ++ Fs_+, Cs ++ Cs_+, Rs ++ Rs_+, Ps ++ Ps_+)$$

where  $++$  denotes stack concatenation.

**Theorem 3.**

$$\begin{array}{c} (H, Fs, Cs, Rs, Ps) \longrightarrow (H', Fs', Cs', Rs', Ps') \\ \Downarrow \\ \frac{(H, Fs, Cs, Rs, Ps)}{\Lambda} \longrightarrow \frac{(H', Fs', Cs', Rs', Ps')}{\Lambda} \end{array}$$

*Proof.* We consider each reduction rule in turn that does not lead to FAIL.

– E-LOCAL

$$\begin{array}{c} (H, F \circ Fs, Cs, Rs, (\mathbf{local} \ x:T; \bar{s}) \circ Ps) \\ \downarrow \\ (H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{array}$$

If we take  $\Lambda$  to be  $(Fs_+, Cs_+, Rs_+, Ps_+)$ , then

$$\frac{(H, F \circ Fs, Cs, Rs, (\mathbf{local} \ x:T; \bar{s}) \circ Ps)}{\Lambda}$$

is equal to

$$(H, (F \circ Fs) ++ Fs_+, Cs ++ Cs_+, Rs ++ Rs_+, ((\mathbf{local} \ x:T; \bar{s}) \circ Ps) ++ Ps_+)$$

which is equal to

$$(H, F \circ (Fs ++ Fs_+), Cs ++ Cs_+, Rs ++ Rs_+, (\mathbf{local} \ x:T; \bar{s}) \circ (Ps ++ Ps_+))$$

According to E-LOCAL

$$\begin{array}{c} (H, F \circ (Fs ++ Fs_+), Cs ++ Cs_+, Rs ++ Rs_+, (\mathbf{local} \ x:T; \bar{s}) \circ (Ps ++ Ps_+)) \\ \downarrow \\ (H, (F, x \mapsto \mathbf{null}) \circ (Fs ++ Fs_+), Cs ++ Cs_+, Rs ++ Rs_+, \bar{s} \circ (Ps ++ Ps_+)) \end{array}$$

where the reached state is equal to

$$(H, (F, x \mapsto \mathbf{null}) \circ Fs ++ Fs_+, Cs ++ Cs_+, Rs ++ Rs_+, (\bar{s} \circ Ps) ++ Ps_+)$$

which is equal to

$$\frac{(H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps)}{\Lambda}$$

Hence, we have

$$\begin{aligned} & \frac{(H, F \circ Fs, Cs, Rs, (\mathbf{local } x:T; \bar{s}) \circ Ps)}{\Lambda} \\ & \quad \downarrow \\ & \frac{(H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps)}{\Lambda} \end{aligned}$$

which is what we needed to prove.

– E-STORENULL

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x=\mathbf{null}; \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

We lift the former state:

$$\frac{(H, F \circ Fs, Cs, Rs, (x=\mathbf{null}; \bar{s}) \circ Ps)}{(Fs_+, Cs_+, Rs_+, Ps_+)}$$

which is equal to

$$(H, (F \circ Fs) ++ Fs_+, Cs ++ Cs_+, Rs ++ Rs_+, ((x=\mathbf{null}; \bar{s}) \circ Ps) ++ Ps_+)$$

which is equal to

$$(H, F \circ (Fs ++ Fs_+), Cs ++ Cs_+, Rs ++ Rs_+, (x=\mathbf{null}; \bar{s}) \circ (Ps ++ Ps_+))$$

which, according to E-STORENULL, reaches

$$(H, (F, x \mapsto \mathbf{null}) \circ (Fs ++ Fs_+), Cs ++ Cs_+, Rs ++ Rs_+, \bar{s} \circ (Ps ++ Ps_+))$$

which is equal to

$$(H, ((F, x \mapsto \mathbf{null}) \circ Fs) ++ Fs_+, Cs ++ Cs_+, Rs ++ Rs_+, (\bar{s} \circ Ps) ++ Ps_+)$$

which is equal to

$$\frac{(H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps)}{(Fs_+, Cs_+, Rs_+, Ps_+)}$$

Hence,

$$\frac{(H, F \circ Fs, Cs, Rs, (x=\mathbf{null}; \bar{s}) \circ Ps)}{(Fs_+, Cs_+, Rs_+, Ps_+)}$$

↓

$$\frac{(H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps)}{(Fs_+, Cs_+, Rs_+, Ps_+)}$$

which is what we needed to prove.

- E-STOREFIELD: analogous.
- E-STORENEW: analogous.
- E-STOREWRITEFIELD: analogous.
- E-WRITEFIELD: analogous.
- E-IFTRUE: analogous.
- E-IFFALSE: analogous.
- E-METHODCALL: analogous.
- E-EXITMETHOD. We deal with this case explicitly as it is the only rule which removes items from the stacks. We are given

$$\begin{aligned} & (H, F' \circ F \circ Fs, C \circ Cs, x \circ Rs, \epsilon \circ Ps) \\ & \quad \downarrow \\ & (H, (F, x \mapsto F'[\mathbf{result}]) \circ Fs, Cs, Rs, Ps) \end{aligned}$$

where

$$\begin{aligned} R &= x \circ R' \\ C(H, F'[\mathbf{result}]) &= \mathit{true} \end{aligned}$$

We write the initial state as:

$$\frac{(H, F' \circ F \circ Fs, C \circ Cs, x \circ Rs, \epsilon \circ Ps)}{(Fs_+, Cs_+, Rs_+, Ps_+)}$$

which is equal to

$$(H, (F' \circ F \circ Fs) ++ Fs_+, (C \circ Cs) ++ Cs_+, (x \circ Rs) ++ Rs_+, (\epsilon \circ Ps) ++ Ps_+)$$

which is equal to

$$(H, F' \circ F \circ (Fs ++ Fs_+), C \circ (Cs ++ Cs_+), x \circ (Rs ++ Rs_+), \epsilon \circ (Ps ++ Ps_+))$$

We can apply E-EXITMETHOD since we know the necessary condition (i.e.  $C(H, F'[\mathit{result}])$ ) holds. Thus, the state single-steps to

$$(H, (F, x \mapsto F'[\mathbf{result}]) \circ (Fs ++ Fs_+), Cs ++ Cs_+, Rs ++ Rs_+, Ps ++ Ps_+)$$

which is equal to

$$(H, ((F, x \mapsto F'[\mathbf{result}]) \circ Fs) ++ Fs_+, Cs ++ Cs_+, Rs ++ Rs_+, Ps ++ Ps_+)$$

which is equal to

$$\frac{(H, (F, x \mapsto F'[\mathbf{result}]) \circ Fs, Cs, Rs, Ps)}{(Fs_+, Cs_+, Rs_+, Ps_+)}$$

Hence

$$\frac{(H, F' \circ F \circ Fs, C \circ Cs, x \circ Rs, \epsilon \circ Ps)}{(Fs_+, Cs_+, Rs_+, Ps_+)}$$

↓

$$\frac{(H, ((F, x \mapsto F'[\mathbf{result}]) \circ Fs), Cs, Rs, Ps)}{(Fs_+, Cs_+, Rs_+, Ps_+)}$$

which is what we needed to show. □

**Theorem 4.**

$$(H, Fs, Cs, Rs, Ps) \longrightarrow \text{FAIL} \Rightarrow \frac{(H, Fs, Cs, Rs, Ps)}{\Lambda} \longrightarrow \text{FAIL}$$

*Proof.* By considering each reduction leading to failure in turn. □

**Theorem 5.** *If*

$$(H, Fs, Cs, Rs, Ps) \longrightarrow^* (H, Fs, Cs', Rs', Ps')$$

*then*

$$\frac{(H, Fs, Cs, Rs, Ps)}{\Lambda} \longrightarrow^* \frac{(H, Fs, Cs', Rs', Ps')}{\Lambda}$$

*and, if*

$$(H, Fs, Cs, Rs, Ps) \longrightarrow^* \text{FAIL}$$

*then*

$$\frac{(H, Fs, Cs, Rs, Ps)}{\Lambda} \longrightarrow^* \text{FAIL}$$

*Proof.* Follows directly from Theorem 3 and Theorem 4. □

**Definition 13 (Final state).** *We define a final state as follows:*

$$\text{final}(\text{FAIL})$$

*and*

$$\forall H, F \bullet \text{final}(H, F \circ \epsilon, \epsilon, \epsilon, \epsilon \circ \epsilon)$$

**Definition 14 (Well-formed state).** *We say a state  $\sigma = (H, Fs, Cs, Rs, Ps)$  is well-formed, written  $\text{wf}(\sigma)$  iff*

$$|Fs| = |Cs| + 1 = |Rs| + 1 = |Ps|$$

*We also consider FAIL to be well-formed.*

We show that the single step rule preserves the well-formedness of a state, i.e. if a state is well-formed, all applicable reduction rules will lead to another well-formed state.

**Theorem 6 (State well-formedness preservation).**

$$\forall \sigma, \sigma' \bullet \text{wf}(\sigma) \Rightarrow \sigma \longrightarrow \sigma' \Rightarrow \text{wf}(\sigma')$$

*Proof.* We consider each reduction rule in turn. It is trivial to see that, given  $\sigma \longrightarrow \sigma'$  and  $\text{wf}(\sigma)$ , if the applied reduction rule does not push or pop items from any of the state's stack,  $\sigma'$  will satisfy the conditions mentioned in Definition 14 and thus be well-formed.

– E-LOCAL

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (\mathbf{local} \ x:T; \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

No stack changes.

– E-STORENULL

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x=\mathbf{null}; \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

No stack changes.

– E-STOREFIELD

$$(H, F \circ Fs, Cs, Rs, (x=y.f; \bar{s}) \circ Ps) \longrightarrow (H, F' \circ Fs, Cs, Rs, \bar{s} \circ Ps)$$

No stack changes.

– E-STOREFIELDNULL

$$(H, F \circ Fs, Cs, Rs, (x=y.f; \bar{s}) \circ Ps) \longrightarrow \text{FAIL}$$

Trivial as FAIL is well-formed by definition.

– E-STORENEW

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x=\mathbf{new} \ K; \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H', (F', x \mapsto \mathit{oid}) \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

No stack changes.

– E-WRITEFIELD

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x.f = y; \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H', F \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

No stack changes.

- E-WRITEFIELDNULL

$$(H, F \circ Fs, Cs, Rs, x.f = y; \bar{s} \circ Ps) \longrightarrow \text{FAIL}$$

Trivial as FAIL is well-formed by definition.

- E-IFTRUE

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (\text{if } (x == y) s_1 \text{ else } s_2 \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F \circ Fs, Cs, Rs, (s_1 \bar{s}) \circ Ps) \end{aligned}$$

No stack changes.

- E-IFFALSE

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (\text{if } (x == y) s_1 \text{ else } s_2 \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F \circ Fs, Cs, Rs, (s_2 \bar{s}) \circ Ps) \end{aligned}$$

No stack changes.

- E-METHODCALL

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x=y.T::m(\bar{z}) \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F' \circ F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs, body_{T::m} \circ \bar{s} \circ Ps) \end{aligned}$$

Each stack contains an extra item, so it is easy to see that if  $\text{wf}(\sigma)$ , then also  $\text{wf}(\sigma')$ .

- E-METHODCALLFAIL

$$(H, F \circ Fs, Cs, Rs, x=y.T::m(\bar{z}) \bar{s} \circ Ps) \longrightarrow \text{FAIL}$$

Trivial as FAIL is well-formed by definition.

- E-METHODCALLNULL

$$(H, F \circ Fs, Cs, Rs, x=y.T::m(\bar{z}) \bar{s} \circ Ps) \longrightarrow \text{FAIL}$$

Trivial as FAIL is well-formed by definition.

- E-EXITMETHOD

$$\begin{aligned} & (H, F' \circ F \circ Fs, C \circ Cs, x \circ Rs, \epsilon \circ Ps) \\ & \quad \downarrow \\ & (H, (F, x \mapsto F'[\mathbf{result}]) \circ Fs, Cs, Rs, Ps) \end{aligned}$$

Each stack has one item popped off, so it is clear that if  $\text{wf}(\sigma)$ , then also  $\text{wf}(\sigma')$ .

- E-EXITMETHODFAIL

$$(H, F \circ Fs, C \circ Cs, Rs, \epsilon \circ \bar{s} \circ Ps) \longrightarrow \text{FAIL}$$

Trivial as FAIL is well-formed by definition.

□

**Theorem 7.**

$$\forall \sigma, \sigma' \bullet \text{wf}(\sigma) \Rightarrow \sigma \longrightarrow^* \sigma' \Rightarrow \text{wf}(\sigma')$$

*Proof.* Follows from Theorem 6. □

**Theorem 8.** For any program  $P$ , the initial state  $\sigma_{\text{init}}(P)$  is well-formed.

*Proof.* We know from Definition 9 that

$$\sigma_{\text{init}}(P) = (\epsilon, \epsilon \circ \epsilon, \epsilon, \epsilon, \bar{s} \circ \epsilon)$$

We compute the length of each stack:

$$\begin{aligned} |Fs| &= |\epsilon \circ \epsilon| = 1 \\ |Cs| &= |\epsilon| = 0 \\ |Rs| &= |\epsilon| = 0 \\ |Ps| &= |\bar{s} \circ \epsilon| = 1 \end{aligned}$$

These lengths clearly satisfy the constraints imposed by Definition 14. □

**Definition 15 (Stuck state).** A state is stuck if no reduction rule applies on it.

$$\forall \sigma \bullet (\neg \exists \sigma' \bullet \sigma \longrightarrow \sigma') \Rightarrow \text{stuck}(\sigma)$$

**Theorem 9.** If a state  $\sigma$  is well-formed and the statements in its statement stack well-formed (i.e. come from a well-typed program), then

$$\text{stuck}(\sigma) \iff \text{final}(\sigma)$$

*Proof.* We split the proof in two parts:

- $\text{stuck}(\sigma) \Rightarrow \text{final}(\sigma)$ : trivial by considering every reduction rule in turn.
- $\text{final}(\sigma) \Rightarrow \text{stuck}(\sigma)$ : trivial as no reduction rule applies on a final state. □

**Definition 16 (Big step).** We define a binary relation named “big step relation” as follows:

$$\sigma \Downarrow \sigma' \iff \sigma \longrightarrow^* \sigma' \wedge \text{final}(\sigma')$$

**Theorem 10.** If

$$(H, F \circ Fs, Cs, Rs, (x = y.T :: m(\bar{z}); \bar{s}) \circ Ps) \longrightarrow^* (H', F' \circ Fs, Cs, Rs, \bar{s} \circ Ps)$$

where the second state represents the state immediately after returning from the method invocation, then

$$\text{Pre}_m(H, F) \wedge \text{Post}_m(H, F, H', F'[x])$$

*Proof.* That  $Pre_m(H, F)$  holds is trivial: if this were not the case, E-METHOD-CALLFAIL would apply and FAIL would be reached. We now show that the condition  $Post_m(H, F)(H', F'[\mathbf{result}])$  holds. We first apply E-METHODCALL on the first state:

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x = y.T :: m(\bar{z}); \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F' \circ F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs, body_{T::m} \circ \bar{s} \circ Ps) \end{aligned}$$

Consider the following state

$$\sigma = (H, F' \circ \epsilon, \epsilon, \epsilon, body_{T::m} \circ \epsilon)$$

and let

$$\Lambda = (F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs, \bar{s} \circ Ps)$$

Thus

$$(H, F' \circ F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs, body_{T::m} \circ \bar{s} \circ Ps) = \frac{\sigma}{\Lambda}$$

–  $\sigma \Downarrow \text{FAIL}$ : this is impossible, as according to Theorem 5

$$\sigma \Downarrow \text{FAIL} \Rightarrow \frac{\sigma}{\Lambda} \longrightarrow^* \text{FAIL}$$

which contradicts the premises.

–  $\sigma \Downarrow (H', F'' \circ \epsilon, \epsilon, \epsilon, \epsilon \circ \epsilon)$ , in which case

$$\frac{\sigma}{\Lambda} \longrightarrow^* \frac{(H', F'' \circ \epsilon, \epsilon, \epsilon, \epsilon \circ \epsilon)}{\Lambda}$$

where

$$\frac{(H', F'' \circ \epsilon, \epsilon, \epsilon, \epsilon \circ \epsilon)}{\Lambda} = (H', F'' \circ F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs, \epsilon \circ \bar{s} \circ Ps)$$

We thus have

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x = y.T :: m(\bar{z}); \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F' \circ F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs, body_{T::m} \circ \bar{s} \circ Ps) \\ & \quad \downarrow^* \\ & (H', F'' \circ F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs, \epsilon \circ \bar{s} \circ Ps) \\ & \quad \downarrow \\ & (H', (F, x \mapsto F''[\mathbf{result}]) \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

where the last step is done by E-EXITMETHOD: E-EXITMETHODFAIL cannot apply as it would contradict the premises. Since E-EXITMETHOD imposes that

$$C(H', F''[\mathbf{result}])$$

and  $F''[\mathbf{result}] = F'[x]$  we have what we needed to prove.  $\square$

During verification, we are not interested in the actual evaluation of a program, but only in whether or not failure is encountered. For this, we split the program up in pieces, i.e. we consider each method apart. This is what we mean by modularization.

More concretely, we take a method from the program, we evaluate this method method (keeping in mind we only want to detect failures, so we abstract as much as possible away), and check that no failures occur. If we do this for every method in the program and find that no single method fails, the program in its entirety won't fail either. In order to do this, we need to apply some changes:

- We need a new kind of state: as we are confining execution to within a method, we don't need the stacks: only the top items are of interest. Hence, a state  $(H, F \circ Fs, C \circ Cs, Rs, \bar{s} \circ Ps)$  gets “flattened” to  $(H, F, C, \bar{s})$ .
- We also need to know what initial state to use. The only guarantee we have is that the method's precondition will be satisfied, so we have to consider every state which satisfies those as initial state.
- A method body often contains calls to other methods. Since we want to limit ourselves to only one method, we must find a way to deal with those invocations. For this, we completely rely on the called method's specifications. Method invocation then gets dealt with by a new reduction rule, which first checks that the current state satisfies the invoked method's preconditions and then produces a new state which is only guaranteed to satisfy the method's postconditions, meaning this new method-invocation rule is nondeterministic.
- Not only do we want a method execution not to fail when starting off from a random state satisfying the method's preconditions, we also want it to end in a state which satisfies the method's postcondition. In our operational semantics, **E-ExitMethod** takes care of verifying this. However, we can't use this rule directly as it makes execution leave the current method.

We have given these new rules the name “step over relation”, and they are fully defined in Figure 2.

**Definition 17 (Step over relation).** *We introduce a new binary relation over states which we call the step-over relation. The exact rules are contained in Figure 2.*

|                     |  |   |
|---------------------|--|---|
| H-LOCAL             | $(H, F, C, \mathbf{local} \ x:T; \bar{s}) \rightsquigarrow (H, F', C, \bar{s})$  | $F' = F, x \mapsto \mathbf{null}$   |
| H-STORENULL         | $(H, F, C, x=\mathbf{null}; \bar{s}) \rightsquigarrow (H, F', C, \bar{s})$   | $F' = F, x \mapsto \mathbf{null}$   |
| H-STOREFIELD        | $(H, S, C, x=y.f; \bar{s}) \rightsquigarrow (H, S', C, \bar{s})$   | $oid = F[y] \neq \mathbf{null}$<br>$F' = F, x \mapsto H[oid][f]$  |
| H-STOREFIELDNULL    | $(H, F, C, x=y.f; \bar{s}) \rightsquigarrow \mathbf{FAIL}$   | $F[y] = \mathbf{null}$  |
| H-STORENEW          | $(H, F, C, x=\mathbf{new} \ K; \bar{s}) \rightsquigarrow (H', F', C, \bar{s})$   | $F' = F, x \mapsto oid, oid \notin H$<br>$H' = H, oid \mapsto \mathbb{F}$<br>$\mathbb{F} = \epsilon, \bar{f} \mapsto \mathbf{null}$ |
| H-WRITEFIELD        | $(H, S, C, x.f = y; \bar{s}) \rightsquigarrow (H', S, C, \bar{s})$   | $oid = F[x] \neq \mathbf{null}$<br>$\mathbb{F}' = H[oid], f \mapsto F[y]$<br>$H' = H, oid \mapsto \mathbb{F}'$                      |
| H-WRITEFIELDNULL    | $(H, F, C, x.f = y; \bar{s}) \rightsquigarrow \mathbf{FAIL}$   | $F[y] = \mathbf{null}$  |
| H-IFTRUE            | $(H, F, C, \mathbf{if} (x == y) \ s_1 \ \mathbf{else} \ s_2 \ \bar{s})$<br>$\rightsquigarrow (H, F, C, s_1 \ \bar{s})$ | $F[x] = F[y]$   |
| H-IFFALSE           | $(H, F, C, \mathbf{if} (x == y) \ s_1 \ \mathbf{else} \ s_2 \ \bar{s})$<br>$\rightsquigarrow (H, F, C, s_2 \ \bar{s})$ | $F[x] \neq F[y]$  |
| H-METHODCALL        | $(H, F, C, x=y.T::m(\bar{z}) \ \bar{s}) \rightsquigarrow (H', F', C, \bar{s})$   | $oid = F[y] \neq \mathbf{null}$<br>$Pre_{T::m}(H, F) = true$<br>$Post_{T::m}(H, F)(H', S') = true$                                  |
| H-METHODCALLPREFAIL | $(H, F, C, x=y.T::m(\bar{z}) \ \bar{s}) \rightsquigarrow \mathbf{FAIL}$  | $Pre_{T::m}(H, S) \neq true$  |
| H-METHODCALLNULL    | $(H, F, C, x=y.T::m(\bar{z}) \ \bar{s}) \rightsquigarrow \mathbf{FAIL}$  | $F[y] = \mathbf{null}$  |
| H-METHODEXITFAIL    | $(H, F, C, \epsilon) \rightsquigarrow \mathbf{FAIL}$   | $C(H, F) \neq true$   |

**Fig. 2.** The step-over relation

**Definition 18 (Multistep-over relation).** *The multisteps-over relation is defined as follows:*

$$\frac{}{\sigma \rightsquigarrow^* \sigma} \text{H}^*\text{-REFLEXIVE}$$

$$\frac{\sigma_1 \rightsquigarrow \sigma_2 \quad \sigma_2 \rightsquigarrow^* \sigma_3}{\sigma_1 \rightsquigarrow^* \sigma_3} \text{H}^*\text{-STEPOVER}$$

The following theorem is an important step towards modularization: it states that we can replace the method call reduction rule by another nondeterministic

one that just obeys the method’s specifications, and this without losing “failure preservation”, i.e. that if the original rules fail, so will the new ones.

**Theorem 11.** *If we consider the execution of a method body from immediately after E-METHODCALL to just before E-EXITMETHOD and replace E-METHODCALL by a nondeterministic rule that leads to a random state satisfying the invoked method’s postcondition, then, if the step-rules lead to failure, so will the step-over rules.*

*Proof.* We write our nondeterministic variant of the step-relation as  $\sigma \longrightarrow_m \sigma'$ . We first show there exists a relationship between the step-states and the step-over states. We introduce a flattening function:

$$[(H, F \circ Fs, C \circ Cs, Rs, \bar{s} \circ Ps)] = (H, F, C, Ps) \quad [\text{FAIL}] = \text{FAIL}$$

It is easy to show that if  $\sigma \longrightarrow_m \sigma'$ , then  $[\sigma] \rightsquigarrow [\sigma']$ . From this follows that if  $\sigma \longrightarrow_m^* \sigma'$ , then  $[\sigma] \rightsquigarrow^* [\sigma']$ . Now we only need to prove that if  $\sigma \longrightarrow_m \text{FAIL}$ , then  $[\sigma] \rightsquigarrow \text{FAIL}$ . This follows directly from comparing each step rule with its corresponding step-over rule and noting that they behave exactly the same.  $\square$

### 3.4 Soundness of the compilation

Compilation of the source program to BoogiePL<sup>b</sup> is done per method: for each method in the source program, a BoogiePL<sup>b</sup> program is generated according to the following compilation scheme.

The first part of the BoogiePL<sup>b</sup> program – called the prelude and shown in Figure 3 — declares the heap as a finite map mapping (object reference, field name) pairs to values. The alloc field indicates whether an object is allocated or not. The predicates wellformed and successor express properties of heaps: in a wellformed heap fields of allocated objects point to allocated objects, and a heap is a successor of another heap if it is wellformed and contains at least the same objects. Then each method is translated according the schema shown in Figure 4.

The compilation of statements is then straightforward. We define the compilation in full in Figure 5. A method call for instance is compiled to an assert that the receiver of the call is non-null, and that the precondition holds. Then, the heap is havoced (a sound overapproximation of the effects the method has on the heap), and then it is assumed that the new heap is a successor of the old heap (according to the successor predicate defined in the prelude) and that the postcondition of the call holds.

The key soundness theorem for this compilation is the following: if the method can fail under the step-over relation when started in a state satisfying its precondition, then the compiled BoogiePL<sup>b</sup> program can also fail.

**Theorem 12.** *Given a program  $P$ , let  $m$  be a method of  $P$  and  $c$  its BoogiePL<sup>b</sup> translation (see Figures 4 and 5). If for any  $H, F$  for which  $\text{Pre}_m(H, F)$  is true,*

$$(H, F, \text{Post}_m(H, F), \text{body}_m) \rightsquigarrow^* \text{FAIL}$$

```

var heap : [ref, name]any;
const alloc : name;

function wellformed([ref, name]any) returns (bool);
axiom ( $\forall h : [\text{ref}, \text{name}]any \bullet \text{wellformed}(h) \Rightarrow$ 
 $(\forall r : \text{ref}, f : \text{name} \bullet r \neq \text{null} \wedge h[r, \text{alloc}] \Rightarrow h[r, f] = \text{null} \vee h[h[r, f], \text{alloc}])$ )

function successor([ref, name]any, [ref, name]any) returns (bool);
axiom ( $\forall \text{before} : [\text{ref}, \text{name}]any, \text{after} : [\text{ref}, \text{name}]any \bullet$ 
 $\text{successor}(\text{before}, \text{after}) \Rightarrow$ 
 $\text{wellformed}(\text{after}) \wedge (\forall r : \text{ref} \bullet \text{before}[r, \text{alloc}] \Rightarrow \text{after}[r, \text{alloc}]))$ );

const f : name;    for every field f

```

**Fig. 3.** Prelude of the BoogiePL<sup>b</sup> program

```

var  $id_{local}$  : ref;    for every declared local variable in the method
var initheap : [ref, name]any;
initheap := heap;
var  $id_{arg}$ ;
var  $init\$id_{arg}$ ;
 $init\$id_{arg} := id_{arg}$ ; }    for every argument
assume this  $\neq$  null;
assume wellformed(heap);
assume  $Pre_{T::m}$ ;
Translation of the method body
assert  $Post_{T::m}$ 

```

**Fig. 4.** Translation of a method  $m$

then there exists a store  $\mu$

$$(c \mid \mu) \rightsquigarrow^* \mathbf{fail}$$

*Proof.* We introduce the notion of “synchronized states”: a state from the step-over semantics is synchronized with a BoogiePL<sup>b</sup> state if the heap and local variables are bound to equivalent values in both languages. We use the following notation: if  $\sigma$  is the state in the step-over evaluation (i.e. a quadruple  $(H, F, C, \bar{s})$  or FAIL),  $[\sigma]$  denotes synchronized the BoogiePL<sup>b</sup> state (i.e. of the form  $(c \mid mu)$ , or **fail**, respectively).

When applying a step-over rule, we can apply a number of corresponding BoogiePL<sup>b</sup> steps so that we end up with a synchronized state:

$$\sigma \rightsquigarrow \sigma' \Rightarrow [\sigma] \rightsquigarrow^* [\sigma']$$

Note that a single step-over step must be matched by one or more BoogiePL<sup>b</sup> steps. The actual number of steps is primarily determined by the translation, i.e.

|  |   |
|--|---|
| <b>local</b> x : T;  | x := <b>null</b>  |
| x = <b>null</b> ;  | x := <b>null</b>  |
| x = y.f;   | <b>assert</b> y ≠ <b>null</b> ;<br>x := heap[y, f]  |
| x = y.m( $\bar{z}$ )   | {<br><b>var</b> oldheap : [name, ref]any;<br>oldheap := heap;<br><b>assert</b> y ≠ <b>null</b> ;<br><b>assert</b> $Pre_m$ ;<br><b>havoc</b> heap;<br><b>assume</b> successor(oldheap, heap);<br><b>assume</b> $Post_m$<br>}                                     |
| x = <b>new</b> T;  | {<br><b>var</b> oid : ref;<br><b>assume</b> oid ≠ <b>null</b> ;<br><b>assume</b> heap[oid, alloc] = <b>false</b> ;<br><b>assume</b> ( $\forall f : \mathbf{name} \bullet \text{heap}[\text{oid}, f] = \mathbf{null}$ );<br>heap[oid, alloc] := <b>true</b><br>} |
| x.f = y;   | <b>assert</b> x ≠ <b>null</b> ;<br>heap[x, f] := y  |
| <b>if</b> ( x == y ) s <sub>1</sub> <b>else</b> s <sub>2</sub> | {<br><b>assume</b> x = y;<br><i>translation of s<sub>1</sub></i><br>[]<br><b>assume</b> x ≠ y;<br><i>translation of s<sub>2</sub></i><br>}  |

Fig. 5. Compilation scheme

to how many BoogiePL<sup>b</sup> commands a single toy-language statement is compiled. Also note that on the BoogiePL<sup>b</sup> side, we only need to reach a synchronized state by following some path: the nondeterministic BoogiePL<sup>b</sup> evaluation might also lead to many other nonrelated states.

The first few lines of the translation, as shown in Figure 4, contain variable declarations and initializations and require no further explanation. The three assumptions are not strictly necessary, but (drastically) lower the number of false negatives. They are explained in more detail in a later paragraph. From here on, there exists a BoogiePL<sup>b</sup> state which is synchronized with the step-over state at the beginning of the method.

Next comes the statement translation followed by the assertion of the method's postconditions. We now show that this series of BoogiePL<sup>b</sup> commands match the corresponding toy language statements. We do this by considering the different forms a method body can take:

- **local**  $x : T$ ;  $\bar{s}$ : trivial. The corresponding BoogiePL<sup>b</sup> local variable is set to **null**, and there is no step-over rule leading to failure.
- $x = \mathbf{null}$ ;  $\bar{s}$ : trivial.
- $x = y.f$ ;  $\bar{s}$ : either H-STOREFIELD and H-STOREFIELDNULL apply. Rule H-STOREFIELDNULL leads to failure, hence so must the translation. This rule applies when  $F[y] = \mathbf{null}$  and the translation contains a **assert**  $y \neq \mathbf{null}$ , meaning that the BoogiePL<sup>b</sup> translation will also fail in matching circumstances. H-STOREFIELD fetches a value from the heap and stores it in a local variable which is taken care of by the rest of the translation. Thus, regardless of whether H-STOREFIELD or H-STOREFIELDNULL applies, in both cases the states stay synchronized.
- $x = \mathbf{new} T$ ;  $\bar{s}$ : only H-STORENEW applies. The trickiest part here is to translate the uniqueness of *oid*. We do this by introducing a phantom field *alloc* which indicates whether an object exists or not. If it does not exist, the *oid* referring to it must be free for use. So, the generation of a unique *oid* in BoogiePL<sup>b</sup> happens by taking a random *oid*, making sure no object exists with this *oid* (**assert**  $\text{heap}[\text{oid}, \text{alloc}] = \mathbf{false}$ ) and immediately setting the *alloc* field to true, making sure that the same *oid* will not be used again later.
- $x.f = y$ ;  $\bar{s}$ : the translation needs to match both H-WRITEFIELD and H-WRITEFIELDNULL. The latter applies when  $x$  is bound to **null** and leads to failure: the **assert**  $x \neq \mathbf{null}$  command takes care of this part of the synchronization. The behaviour of H-WRITEFIELD is matched by the BoogiePL<sup>b</sup> heap updating command.
- **if** (  $x == y$  )  $s_1$  **else**  $s_2$   $\bar{s}$ : either H-IFTRUE or H-IFFALSE can apply. Both cases are dealt with by using a BoogiePL<sup>b</sup> choice command, each branch representing a possible execution flow.
- $\epsilon$ : either the state is stuck or H-METHODEXITFAIL applies. We don't need to worry about the former case, but we have to ensure that failure will occur in similar circumstances: this is done by the final **assert** command (see Figure 4) which will make the BoogiePL<sup>b</sup> execution fail if the postcondition happens not to be true at the end of the method.
- $x = y.m(\bar{z})$ ;  $\bar{s}$ : first a copy of the heap is made so as to be able to refer to it in the last line (the postcondition assertion). The next line, **assert**  $y \neq \mathbf{null}$ , takes care of simulating **H-MethodCallNull**, while **assert**  $Pre_m$  deals with **H-MethodCallPreFail**. After a method returns, it might have changed any part of the heap, so we erase any information we have about it with the **havoc** statement. However, as explained in the following paragraph, we know that the new heap is a successor (Definition 21) to the old heap (Theorem 16), and that the postconditions will apply (Theorem 10). These two assumptions can be made and help us lower the number of false negatives.

□

### 3.5 Auxiliary axioms

Our verifier promises that if it accepts a program, said program will never encounter failure during its execution. It does however not make any guarantees

regarding a program's success: if a program does not verify, it means it could fail, but might as well run correctly. In short, false positives (the program verifies but fails at runtime) are excluded, but false negatives (the program does not verify, but will not fail at runtime) are not. To maximize a verifier's usability, we wish to minimize the number of false negatives. For this, we can provide BoogiePL<sup>b</sup> with more information about the heap in the form of axioms.

**Definition 19 (Well-formed heap).** *We say the heap is well-formed, written  $\text{wf}(H)$  if for each object it contains the fields are either **null** or refer to other existent objects.*

$$\text{wf}(H) \iff (\forall \text{oid}, \text{oid}', f \bullet \text{oid}' = H[\text{oid}][f] \Rightarrow \text{oid}' = \mathbf{null} \vee \text{oid}' \in \text{dom}(H))$$

**Definition 20 (Well-formed frame stack).** *We say that the frame stack is well-formed with respect to a heap  $H$ , written  $\text{wf}(Fs, H)$  if it binds local variables either to **null** or to an oid element of the heap  $H$ 's domain.*

$$\text{wf}(F \circ Fs, H) = (\forall x, \text{oid} \bullet \text{oid} = F[x] \Rightarrow \text{oid} \in \text{dom}(H)) \wedge \text{wf}(Fs, H) \\ \text{wf}(\epsilon, H)$$

**Theorem 13.** *The single-step relation preserves the well-formedness of the states' heaps as well as the well-formedness of stack frames with respect to the heap of the same state. Formally, if*

$$\text{wf}(H) \wedge \text{wf}(Fs, H) \wedge (H, Fs, Cs, Rs, Ps) \longrightarrow (H', Fs', Cs', Rs', Ps')$$

then

$$\text{wf}(H') \wedge \text{wf}(Fs', H')$$

*Proof.* We consider each reduction rule which doesn't lead to failure in turn:

– E-LOCAL

$$(H, F \circ Fs, Cs, Rs, (\mathbf{local} \ x:T; \bar{s}) \circ Ps) \\ \downarrow \\ (H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps)$$

The heap remains unchanged and the upper stack frame is extended with a binding to **null**, both of which preserve the well-formedness of the heap and the frame stack.

– E-STORENULL

$$(H, F \circ Fs, Cs, Rs, (x=\mathbf{null}; \bar{s}) \circ Ps) \\ \downarrow \\ (H, (F, x \mapsto \mathbf{null}) \circ Fs, Cs, Rs, \bar{s} \circ Ps)$$

The heap remains unchanged and the upper stack frame is modified by rebinding a variable to **null**, both of which preserve the well-formedness of the heap and the frame stack.

## – E-STOREFIELD

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x=y.f; \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F' \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

where

$$\begin{aligned} oid &= F[y] \neq \mathbf{null} \\ F' &= F, x \mapsto H[oid][f] \end{aligned}$$

The heap remains unchanged, so its well-formedness is clearly preserved. We know that  $oid \in \text{dom}(H)$  as it is guaranteed by  $\text{wf}(F \circ Fs, H)$ . From  $\text{wf}(H)$  follows that  $H[oid][f] \in \text{dom}(H)$ . This means a binding to a valid reference  $oid$  is added to the frame stack, thus it remains well-formed with respect to  $H$ .

## – E-STORENEW

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x=\mathbf{new} K; \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H', (F', x \mapsto oid) \circ Fs, Cs, Rs, \bar{s} \circ Ps) \end{aligned}$$

where

$$\begin{aligned} oid &\notin H \\ H' &= H, oid \mapsto \mathbb{F} \\ \mathbb{F} &= \epsilon, f \mapsto \mathbf{null} \end{aligned}$$

The new heap  $H'$  is well-formed as the new object contains only **null** references. The frame stack is well-formed as  $H'$  does contain a mapping for the newly created object.

## – E-WRITEFIELD

$$(H, F \circ Fs, Cs, Rs, (x.f = y; \bar{s}) \circ Ps) \longrightarrow (H', F \circ Fs, Cs, Rs, \bar{s} \circ Ps)$$

where

$$\begin{aligned} oid &= F[y] \neq \mathbf{null} \\ \mathbb{F} &= H[oid], v = F[y] \\ \mathbb{F}' &= \mathbb{F}, f \mapsto v \\ H' &= H, oid \mapsto \mathbb{F}' \end{aligned}$$

The frame stack remains unchanged, thus it remains well-formed. An object's field is changed to a new  $oid$ , which originates from the well-formed stack frame, meaning  $oid \in \text{dom}(H)$ , hence  $H'$  is well-formed.

## – E-IFTRUE

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (\mathbf{if} (x == y) s_1 \mathbf{else} s_2 \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F \circ Fs, Cs, Rs, (s_1 \bar{s}) \circ Ps) \end{aligned}$$

where

$$F[x] = F[y]$$

Neither the heap or frame stack are modified.

– E-IFFALSE

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (\text{if } (x == y) s_1 \text{ else } s_2 \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F \circ Fs, Cs, Rs, (s_2 \bar{s}) \circ Ps) \end{aligned}$$

where

$$F[x] \neq F[y]$$

Neither the heap or frame stack are modified.

– E-METHODCALL stack frame is created containing bindings for the **this** reference, method arguments and **result** variable. The method's postcondition is pushed on the condition stack with the before-state already filled in. The receiver variable (the one to which the method result will be written to upon return) is pushed on the receiver stack.

$$\begin{aligned} & (H, F \circ Fs, Cs, Rs, (x=y.T::m(\bar{z}) \bar{s}) \circ Ps) \\ & \quad \downarrow \\ & (H, F' \circ F \circ Fs, Post_{T::m}(H, F) \circ Cs, x \circ Rs, body_{T::m} \circ \bar{s} \circ Ps) \end{aligned}$$

where

$$\begin{aligned} F &= \epsilon, \mathbf{this} \mapsto oid, \overline{args_{T::m}} \mapsto F[\bar{z}], \mathbf{result} \mapsto \mathbf{null} \\ oid &= F[y] \neq \mathbf{null} \\ Pre_{T::m}(H, F) &= true \end{aligned}$$

The heap remains unchanged. The stack frame is extended with a new frame. The **this** reference and arguments are bound to *oids* which originate from the well-formed frame stack  $F \circ Fs$ , and **result** is bound to **null**, meaning all *oids* are valid (i.e. are elements of the heap's domain).

– E-EXITMETHOD

$$\begin{aligned} & (H, F' \circ F \circ Fs, C \circ Cs, x \circ Rs, \epsilon \circ Ps) \\ & \quad \downarrow \\ & (H, (F, x \mapsto F'[\mathbf{result}]) \circ Fs, Cs, Rs, Ps) \end{aligned}$$

where

$$C(H, F'[\mathbf{result}]) = true$$

The heap remains unchanged, and no new *oids* are introduced on the stack frame. □

**Theorem 14.** *The multistep preserves the well-formedness of heaps.*

$$wf(H) \Rightarrow (H, Fs, Cs, Rs, Ps) \longrightarrow^* (H', Fs', Cs', Rs', Ps') \Rightarrow wf(H')$$

*Proof.* Follows from Theorem 13. □

**Definition 21 (Successor heap).** *We say a heap  $H'$  succeeds a heap  $H$  iff*

$$wf(H') \wedge \forall oid \bullet oid \in \text{dom}(H) \Rightarrow oid \in \text{dom}(H')$$

**Lemma 5.** *If a heap  $H$  is well-formed, it is its own successor.*

$$\forall H \bullet \text{wf}(H) \Rightarrow H \text{ succeeds } H$$

*Proof.* Trivial. □

**Theorem 15.** *If  $H$  is well-formed, and*

$$(H, Fs, Cs, Rs, Ps) \longrightarrow (H', Fs', Cs', Rs', Ps')$$

*then  $H'$  succeeds  $H$ .*

*Proof.* Theorem 13 already shows that well-formedness is preserved by the single step. This leaves us with proving that every *oid* in  $\text{dom}(H)$  is also in  $\text{dom}(H')$ . We can do this easily by considering every reduction rule in turn and show that none of them removes mappings from the heap. □

**Theorem 16.** *If  $H$  is well-formed, and*

$$(H, Fs, Cs, Rs, Ps) \longrightarrow^* (H', Fs', Cs', Rs', Ps')$$

*then  $H'$  succeeds  $H$ .*

*Proof.* Follows directly from Theorem 15. □

## 4 Related and Future Work

There is a huge amount of related work in program verification and hence, we necessarily focus on research results most closely related to ours. For a good overview of the state-of-the-art in program verification for OO languages in general, we refer to [9].

A first closely related line of work is the  $\text{Spec}^\sharp$  verifier [5], one of the state-of-the-art automatic verifiers for Java-like languages. This paper is a first step towards a formal machine-checked soundness proof of the  $\text{Spec}^\sharp$  verifier. So far, only certain key elements of this verifier have been shown sound on paper [3].

The Why/Krakatoa/Caduceus line of tools [7] is a very interesting competitor to the Boogie/ $\text{Spec}^\sharp$ /VCC line of tools: both toolsets are built around a similar intermediate verification language and provide front-ends for Java-like and C-like languages. To avoid trust in the tool-chain, the Why VC generator adopts an approach where the proof produced by Why (possibly with the help of other tools or the user) is checked a posteriori by an automatic checker.

In the Mobius project [1], machine-checked proofs of many programming language properties are being studied with the purpose of supporting proof-carrying code to certify security-related properties of programs. The project has worked out a machine-checked proof of the soundness of their program logic (the Mobius Base Logic) with respect to an operational semantics of Java bytecode. As part of the Mobius project, Lehner and Müller have shown the translation of Java bytecode to BoogiePL sound [10]. This can be seen as another instance of a phase (2) soundness proof in the terminology of this paper.

This paper provides only a first step towards a machine-checked soundness proof of a  $\text{Spec}^\sharp$ -like verifier. In future work, we intend to work out a phase (2) soundness proof for several variants of  $\text{Spec}^\sharp$ -like verifiers.

## 5 Conclusion

BoogiePL<sup>b</sup> is an intermediate language developed to assist the development of verifying compilers by simplifying the generation of verification conditions (VC) ([11]). Unfortunately, no formal soundness proofs existed for this language. To remedy this, this paper provides a full formalization for BoogiePL<sup>b</sup> by defining its syntax, operational semantics (Section 2.2) and VC generation algorithm (Section 2.3). This paper also contains a proof of the VC generation algorithm's soundness with respect to the operational semantics; a full machine-checked proof has been developed and can be found at [2]. Next we have shown how soundness proofs of verifiers that translate to BoogiePL<sup>b</sup> can be proven sound by proving the correctness of their compilation scheme.

## References

1. <http://mobius.inria.fr>.
2. <http://www.cs.kuleuven.be/~frederic/papers/boogie/boogie.v8>.
3. Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87, 2005.
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 *FMCO 2005*, volume 4111 of LNCS, pages 364–387. Springer, 2006.
5. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. pages 49–69. Springer, 2004.
6. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
7. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program verification. In *CAV*, pages 173–177, 2007.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
9. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.
10. H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In M. Huisman and F. Spoto, editors, *Bytecode Semantics, Verification, Analysis and Transformation*, *Electronic Notes in Theoretical Computer Science*, 2007.
11. Rustan Leino and Wolfram Schulte. A verifying compiler for a multi-threaded object-oriented language. Marktoberdorf lecture notes, 2007. In Manfred Broy, Johannes Grünbauer, Tony Hoare (eds.). *Software System Reliability and Security*. IOS Press, 2007.
12. Wolfram Schulte, Songtao Xia, Jan Smans, and Frank Piessens. A Glimpse of a Verifying C Compiler Extended Abstract , 2007.
13. Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine checked soundness proof for an intermediate verification language. *Lecture Notes in Computer Science*. Springer, 2009.
14. Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 349–361, New York, NY, USA, 2008. ACM.

## A The Coq proof script

```

Require Import Arith.
Require Import Omega.

Definition decidable (P : Prop) := P \ / ~ P.
Definition decidable_eq (A : Set) :=
  forall x y : A, decidable (x = y).

Ltac introduce id term :=
  (set (id := term); assert (id = term);
   [ trivial | clearbody id ]).

Section induction_on_size_section.
  Variables
    (A : Set)
    (P : A -> Prop)
    (metric : A -> nat).

  Hypothesis H :
    forall x : A,
      (forall y : A, metric y < metric x -> P y) -> P x.

  Definition Pn (n : nat) :=
    forall x : A, metric x < n -> P x.

  Theorem P_0 : Pn 0.
  Proof.
    red.
    intros.
    apply H.
    intros.
    elim (lt_n_0 _ H0).
  Qed.

  Theorem Hn :
    forall n : nat,
      (forall m : nat, m < n -> Pn m) -> Pn n.
  Proof.
    unfold Pn; intros.
    apply H; intros.
    apply (H0 (metric x) H1 y H2).
  Qed.

  Theorem P_lt_n : forall n m : nat, m < n -> Pn m.
  Proof.

```

```

induction n; intros.
elim (lt_n_0 _ H0).
apply Hn; intros.
apply IHn.
omega.
Qed.

```

```

Theorem P_n : forall n : nat, Pn n.
Proof.
  intros.
  apply (P_lt_n (S n) n).
  auto.
Qed.

```

```

Theorem induction_on_size : forall x : A, P x.
Proof.
  assert (H0 := P_n).
  unfold Pn in H0.
  intros.
  apply (H0 (S (metric x)) x).
  auto.
Qed.

```

End induction\_on\_size\_section.

Section boogie.

```

Variable identifier : Set.

Hypothesis identifier_dec_eq : decidable_eq identifier.

Variable abstract_expression : Set.

Variable value : Set.

Variable true : value.

Hypothesis value_eq_dec : decidable_eq value.

Inductive store : Set :=
| store_empty : store
| store_add   : store -> identifier -> value -> store.

Variable evaluate : abstract_expression -> store -> value.

```

```

Inductive expression : Set :=
| exprConjunction : expression -> expression -> expression
| exprImplication : expression -> expression -> expression
| exprAbstract     : abstract_expression -> expression
| exprForAll       : identifier -> expression -> expression
| exprSubstitute   : identifier -> abstract_expression ->
                    expression -> expression
| exprLiteral      : value -> expression.

```

```

Inductive command : Set :=
| cmdChoice : command -> command -> command -> command
| cmdAssert  : abstract_expression -> command -> command
| cmdAssume  : abstract_expression -> command -> command
| cmdHavoc   : identifier -> command -> command
| cmdSet     : identifier -> abstract_expression ->
                    command -> command
| cmdNil     : command.

```

```

Fixpoint weakest_precondition
(c : command) (P : expression) {struct c} : expression :=
match c with
| cmdChoice x y n =>
  exprConjunction
    (weakest_precondition x (weakest_precondition n P))
    (weakest_precondition y (weakest_precondition n P))
| cmdAssert e n =>
  exprConjunction
    (exprAbstract e)
    (weakest_precondition n P)
| cmdAssume e n =>
  exprImplication
    (exprAbstract e)
    (weakest_precondition n P)
| cmdHavoc x n =>
  exprForAll x (weakest_precondition n P)
| cmdSet x e n =>
  exprSubstitute x e (weakest_precondition n P)
| cmdNil      => P
end.

```

```

Fixpoint metric (c : command) {struct c} : nat :=
match c with
| cmdChoice x y n => S (metric x + metric y + metric n)
| cmdAssert e n => S (metric n)
| cmdAssume e n => S (metric n)

```

```

| cmdHavoc x n => S (metric n)
| cmdSet x e n => S (metric n)
| cmdNil => 0
end.

```

```

Fixpoint append (c c' : command) {struct c} : command :=
  match c with
  | cmdChoice x y n => cmdChoice x y (append n c')
  | cmdAssert e n => cmdAssert e (append n c')
  | cmdAssume e n => cmdAssume e (append n c')
  | cmdHavoc x n => cmdHavoc x (append n c')
  | cmdSet x e n => cmdSet x e (append n c')
  | cmdNil => c'
  end.

```

```

Inductive state : Set :=
| in_progress : command -> store -> state
| failure : state.

```

Notation "[ c | s ]" := (in\_progress c s).

Reserved Notation "x ---> y" (at level 50, left associativity).

```

Inductive step : state -> state -> Prop :=
| stepChoiceLeft : forall x y n s,
  [ cmdChoice x y n | s ] ---> [ append x n | s ]
| stepChoiceRight : forall x y n s,
  [ cmdChoice x y n | s ] ---> [ append y n | s ]
| stepAssertTrue : forall e n s,
  evaluate e s = true ->
  [ cmdAssert e n | s ] ---> [ n | s ]
| stepAssertFalse : forall e n s,
  evaluate e s <> true ->
  [ cmdAssert e n | s ] ---> failure
| stepAssume : forall e n s,
  evaluate e s = true ->
  [ cmdAssume e n | s ] ---> [ n | s ]
| stepHavoc : forall x n s v,
  [ cmdHavoc x n | s ] ---> [ n | store_add s x v ]
| stepSet : forall x e n s,
  [ cmdSet x e n | s ] --->
  [ n | store_add s x (evaluate e s) ]

```

where "x ---> y" := (step x y).

Theorem metric\_append :

```

forall c c', metric (append c c') = metric c + metric c'.
Proof.
  induction c; intros; simpl;
  try solve [ rewrite (IHc c'); trivial ].
  rewrite (IHc3 c').
  auto with arith.
  trivial.
Qed.

Theorem step_decreases_command_size :
  forall c c' s s', [ c | s ] ---> [ c' | s' ] -> metric c' < metric c.
Proof.
  intros.
  introduce S1 [c | s].
  introduce S2 [c' | s'].
  rewrite <- H0 in H.
  rewrite <- H1 in H.
  induction H; try (discriminate; fail); injection H0; injection H1;
  intros; subst; subst; simpl; try (rewrite metric_append); omega.
Qed.

Fixpoint state_metric (s : state) {struct s} : nat :=
  match s with
  | in_progress c s => S (metric c)
  | failure          => 0
  end.

Theorem step_decreases_state_size :
  forall s s', s ---> s' -> state_metric s' < state_metric s.
Proof.
  intros.
  destruct s; destruct s'.
  assert (H0 := step_decreases_command_size _ _ _ _ H).
  simpl.
  omega.
  simpl.
  omega.
  inversion H.
  inversion H.
Qed.

Reserved Notation "x --*> y" (at level 50, left associativity).

Inductive mstep : state -> state -> Prop :=
| mstep_reflexive : forall s, s --*> s

```

```

| mstep_step      : forall s s' s'',
                   s ---> s' -> s' --*> s'' -> s ---> s'',
where "x --*> y" := (mstep x y).

```

Definition fails state := state --\*> failure.

Definition succeeds state := (~ (fails state)).

```

Fixpoint is_true (e : expression) (s : store) {struct e} : Prop :=
  match e with
  | exprConjunction x y => is_true x s /\ is_true y s
  | exprImplication x y => is_true x s -> is_true y s
  | exprAbstract e      => evaluate e s = true
  | exprForAll x e      => forall v, is_true e (store_add s x v)
  | exprLiteral v      => v = true
  | exprSubstitute x a e => is_true e (store_add s x (evaluate a s))
  end.

```

Definition induction\_on\_command\_size P :=  
 induction\_on\_size command P metric.

Definition induction\_on\_state\_size P :=  
 induction\_on\_size state P state\_metric.

```

Theorem command_induction
(P : command -> Prop)
(H_nil : P cmdNil)
(H_assert : forall e n, P n -> P (cmdAssert e n))
(H_assume : forall e n, P n -> P (cmdAssume e n))
(H_havoc : forall x n, P n -> P (cmdHavoc x n))
(H_choice : forall a b n,
  P (append a n) -> P (append b n) ->
  P (cmdChoice a b n))
(H_set : forall x e n, P n -> P (cmdSet x e n))
: forall c, P c.

```

Proof.

```

intros.
refine (induction_on_command_size P _ c).
intros.
destruct x; try (first [ apply H_nil | apply H_assert |
  apply H_assume | apply H_havoc |
  apply H_set ]);
  apply H; simpl; omega; fail).
apply H_choice; apply H; simpl; rewrite metric_append; omega.
Qed.

```

```

Lemma wp_impl_assert_n : forall e n s P,
  is_true (weakest_precondition (cmdAssert e n) P) s ->
  is_true (weakest_precondition n P) s.
Proof.
  intros.
  change (weakest_precondition (cmdAssert e n) P) with
    (exprConjunction (exprAbstract e) (weakest_precondition n P)) in H.
  simpl in H.
  destruct H.
  trivial.
Qed.

```

```

Lemma wp_impl_assert_e : forall P e n s,
  is_true (weakest_precondition (cmdAssert e n) P) s ->
  is_true (exprAbstract e) s.
Proof.
  intros.
  change (weakest_precondition (cmdAssert e n) P) with
    (exprConjunction (exprAbstract e) (weakest_precondition n P)) in H.
  simpl in H.
  destruct H.
  simpl.
  trivial.
Qed.

```

```

Theorem wp_append : forall a b P,
  weakest_precondition a (weakest_precondition b P) =
  weakest_precondition (append a b) P.
Proof.
  induction a; intros;
  try (solve [ simpl; f_equal; rewrite IHa; trivial ]).
  simpl.
  f_equal; f_equal; rewrite IHa3; trivial.
Qed.

```

```

Theorem soundness : forall c store,
  is_true (weakest_precondition c (exprLiteral true)) store ->
  succeeds [ c | store ].
Proof.
  apply (command_induction
    (fun c => forall s,
      is_true (weakest_precondition c (exprLiteral true)) s ->
      succeeds [c | s]));
  unfold succeeds; unfold fails; red; intros.

```

```

(* cmdNil *)
inversion H0; subst.
inversion H1.

(* cmdAssert *)
introduce s1 [cmdAssert e n | s]; introduce s2 failure.
rewrite <- H2 in H1; rewrite <- H3 in H1.
destruct H1.
rewrite H2 in H3.
discriminate.
subst s0; subst s''.
inversion H1.
subst e0 n0 s0.
elim (H s).
apply (wp_impl_assert_n _ _ _ _ H0).
subst.
trivial.
subst.
contradiction H7.
assert (H5 := wp_impl_assert_e _ _ _ _ H0).
simpl in H5.
trivial.

(* cmdAssume *)
introduce s1 [cmdAssume e n | s]; introduce s2 failure.
rewrite <- H2 in H1; rewrite <- H3 in H1.
destruct H1.
subst.
discriminate.
subst.
elim (H s).
inversion H1.
subst e0 n0 s0.
generalize H7 H0; clear; intros.
simpl in H0.
apply (H0 H7).
inversion H1; subst.
trivial.

(* cmdHavoc *)
introduce s1 [cmdHavoc x n | s]; introduce s2 failure.
rewrite <- H2 in H1; rewrite <- H3 in H1.
destruct H1.
subst.

```

```

discriminate.
subst.
inversion H1; subst.
elim (H (store_add s x v)).
generalize H0; clear; intros.
simpl in H0.
apply H0.
trivial.

(* cmdChoice *)
introduce s1 [cmdChoice a b n | s]; introduce s2 failure.
rewrite <- H3 in H2; rewrite <- H4 in H2.
destruct H2; subst.
discriminate.
inversion H2; subst; simpl in H1; destruct H1.
elim (H s); [idtac | trivial].
rewrite <- wp_append.
trivial.
elim (H0 s); [idtac | trivial].
rewrite <- wp_append.
trivial.

(* cmdSet *)
introduce s1 [cmdSet x e n | s].
introduce s2 failure.
rewrite <- H2 in H1; rewrite <- H3 in H1.
destruct H1.
subst; discriminate.
subst.
inversion H1; subst.
elim (H (store_add s x (evaluate e s))); [idtac | trivial].
generalize H0; clear; intros.
simpl in H0.
trivial.
Qed.

End boogie.

```