

Inspector Methods for State Abstraction: Soundness Proof

Bart Jacobs and Frank Piessens

Department of Computer Science
Katholieke Universiteit Leuven, Belgium
{bart.jacobs,frank.piessens}@cs.kuleuven.be

May 2, 2007

Abstract

This note formalizes and proves the soundness of an approach for modular static verification of safety properties of object-oriented programs where module specifications refer to module state abstractly, using *inspector methods*, to eliminate dependencies of clients on a module's internal implementation details.

1 Introduction

This note is structured as follows:

- Section 2 defines the syntax and well-formedness rules for a small Java-like programming language with method contracts. Methods are either *inspector methods* or *mutator methods*. Method contracts may include inspector method calls.
- Section 3 defines a dynamic semantics for the language, where execution gets stuck if a language error (such as a null dereference) or a method contract violation occurs.
- Section 4 formalizes an approach for modularly statically verifying that programs do not get stuck. In particular, it defines a program's verification logic and verification conditions, and a notion of *validity* based on provability of the verification conditions in the verification logic.
- Section 5 proves the soundness of the approach, i.e. it proves that valid programs do not get stuck.
- Section 6 provides a number of example programs that are verified by the approach.

2 Programs

We formalize the approach for a dynamically typed Java-like language with interfaces but without subclassing.

We assume the following sets to be given. For each of these sets, the table shows the symbol used to range over the elements of the set.

Set	Element	Meaning
\mathcal{C}	c	class names
\mathcal{I}	I	interface names
\mathcal{F}	f	field names
\mathcal{R}	r	inspector method names
\mathcal{M}	m	mutator method names
\mathcal{X}	x	variable names
\mathcal{OP}	op	operator names

We also assume as given a function $\text{arity} : \mathcal{OP} \rightarrow \mathbb{N}$ that maps an operator name to its arity (i.e., number of arguments).

The syntax of programs and program elements is as follows:

```

program ::= typedecl* main
typedecl ::= class | interface
main ::= s*
class ::= class  $c$  implements  $I^*$ 
        {  $field^*$  rep  $f^*$ ; invariant  $e$ ;  $im^*$  derived_invariant  $e$ ;  $mm^*$  }
interface ::= interface  $I$  {  $imh^*$  dynamic_invariant  $e$ ;  $mmh^*$  }
field ::=  $f$ ;
imh ::= inspector  $r(x^*)$  reads  $x^*$ ; requires  $e$ ;
mmh ::= mutator  $m(x^*)$  requires  $e$ ; modifies  $e^*$ ; ensures  $e$ ;
im ::=  $imh$  { return  $e$ ; }
mm ::=  $mmh$  {  $s^*$  }
g ::= this |  $x$  | result
e ::=  $g$  |  $g_{state}$  |  $op(e^*)$  |  $e.f$  |  $e.f_{state}$  |  $e.inv$  |  $e.committed$  |  $e.r_\tau(e^*)$ 
    |  $e ? e : e$  | forall{ $x$  in  $domain$ ;  $e$ } | heap( $e$ ) |  $e.f_{heap}$  | old( $e$ )
    | checkvirtual( $e$ )
domain ::= ( $e : e$ ) | object | object_statelevel
level ::=  $\tau$  |  $\infty$ 
 $\tau$  ::=  $c$  |  $I$ 
s ::= return  $e$ ; |  $x := e$ ; |  $g.f := g$ ; | call  $x := g.m(g^*)$ ; |  $x := \mathbf{new}$   $c$ ;
    | pack $c$   $g$ ; | unpack $c$   $g$ ; | if ( $e$ ) {  $s^*$  } else {  $s^*$  }

```

The order of the classes and interfaces in the program is significant.

Definition 1. A method A implements a method B if B is declared by an interface I and A is declared by a class c that mentions I in its **implements** clause and the method headers of A and B are identical after substitution of c for I in inspector method call subscripts in the header of B .

Informally, a *valid object state* is one where all applicable invariants hold. A *valid0 object state* is one where **rep** fields hold non-null object references and

the objects pointed to by the **rep** fields are in a valid state. We check statically that in expressions of the form $e.f$ and $e.f_{\text{state}}$, e denotes a valid0 object state, and in inspector method calls, the target expression and argument expressions for parameters mentioned in the **reads** clause denote valid object states.

Definition 2. *A variable occurrence $C[g]$ is valid-stateful if it is an occurrence in the body of an inspector method of **this** or a parameter mentioned in the **reads** clause, or if it is bound by a **object.state**_{level} quantifier, or if it is an occurrence of **this** in a derived invariant or a dynamic invariant.*

*A variable occurrence $C[g]$ is valid0-stateful if it is valid-stateful or it is an occurrence of **this** in an invariant.*

An expression occurrence $C[e]$ is valid-stateful if either it is of the form $C[g_{\text{state}}]$, where $C[g]$ is a valid-stateful variable occurrence, or it is of the form $C[e.f_{\text{state}}]$, where $C[e]$ is a valid0-stateful expression occurrence, or it is of the form $C[\text{heap}(e)]$.

An expression occurrence $C[e]$ is valid0-stateful if it is valid-stateful or it is of the form $C[g_{\text{state}}]$, where $C[g]$ is a valid0-stateful variable occurrence.

We define the relation $R \vdash e : \text{confined}$ inductively in Figure 1.

Note that an expression that contains a subexpression of the form **heap**(e) or $e.f_{\text{heap}}$ is not confined.

Definition 3. *An expression e is a rep expression with respect to a set of expressions R if and only if either e is of the form g_{state} , where g is in R , or e is of the form $e'.f_{\text{state}}$ where e' is a rep expression with respect to R .*

Well-formedness of programs differs from well-formedness in Java in that field and method names must be distinct across classes; that is, they can be thought of as fully-qualified names. Furthermore, well-formedness rules are defined for the constructs that do not have Java counterparts.

Definition 4. *A program is well-formed if*

- *No two classes have the same name. No two interfaces have the same name. No class has the same name as an interface.*
- *The program does not declare two fields with the same name. (I.e., field names must be unique across the whole program.)*
- *The program does not declare two methods with the same name, except if one method implements the other or both methods implement the same method.*
- *Each method name, field name, and type name mentioned in the program is declared by the program. If an inspector method call $e.r_{\tau}(\bar{e})$ appears in the program, then type τ declares an inspector method named r .*
- *No parameter name appears more than once in a given method parameter list.*

$$\begin{array}{c}
R \vdash g : \text{confined} \qquad \frac{R \vdash e_1, \dots, e_n : \text{confined}}{R \vdash \text{op}(e_1, \dots, e_n) : \text{confined}} \qquad \frac{R \vdash e : \text{rep}}{R \vdash e.f : \text{confined}} \\
\\
\frac{R \vdash e : \text{rep}}{R \vdash e.f_{\text{state}} : \text{confined}} \\
\\
\frac{\begin{array}{c} R \vdash e : \text{rep} \quad R \vdash e_1, \dots, e_n : \text{confined} \\ (R \vdash e_i : \text{rep} \text{ for each parameter } x_i \text{ in the reads clause of } m) \end{array}}{R \vdash e.r_\tau(e_1, \dots, e_n) : \text{confined}} \\
\\
\frac{R \vdash e_1, e_2, e_3 : \text{confined}}{R \vdash \text{forall}\{x \text{ in } (e_1 : e_2); e_3\} : \text{confined}} \\
\\
\frac{R \vdash e : \text{confined}}{R \vdash \text{forall}\{x \text{ in object}; e\} : \text{confined}} \\
\\
\frac{R, x \vdash e : \text{confined}}{R \vdash \text{forall}\{x \text{ in object_state}_{\text{level}i}; e\} : \text{confined}} \\
\\
\frac{R \vdash e_1, e_2, e_3 : \text{confined}}{R \vdash (e_1 ? e_2 : e_3) : \text{confined}}
\end{array}$$

Figure 1: Confinedness of an expression

- The field names mentioned in the **rep** clause of a class c are all declared by class c . No field name appears more than once in a given **rep** clause.
- If an inspector method call $e.r_I(\bar{e})$ appears in the contract of a method of interface I , then e is **this_{state}** (if the method is an inspector method) or **heap(this)** (if the method is a mutator method) or of the form

checkvirtual(e')

- The variable names mentioned in a given inspector method's **reads** clause are all in the method's parameter list.
- No variable name appears more than once in a given **reads** clause.
- **old** expressions occur only in **ensures** clauses and they do not occur inside **old** expressions.
- All inspector method bodies are of the form **return** e ; where e is confined to **{this}** plus the parameters in the **reads** clause.
- All invariants, derived invariants, and dynamic invariants are confined to **{this}**
- Expressions of the form $e.inv$ or $e.committed$ appear only in **requires** clauses and **ensures** clauses of mutator methods.
- Quantifications appear only in derived invariants and dynamic invariants, and in **requires** clauses and **ensures** clauses of mutator methods.
- If a class c implements an interface I , then the declaration of I appears after the declaration of c in the program. A call expression $e.r_\tau(\bar{e})$ appears after the declaration of inspector method r in class or interface τ . A quantification over **object_state _{τ}** appears only in the derived invariant or dynamic invariant of type τ . A quantification over **object_state _{∞}** appears only in a mutator method contract.
- Each method body, as well as the program's main routine, contains a return statement as the last statement; and no return statement appears in any other position.
- A **result** expression appears only in a postcondition.
- For each expression occurrence $C[e.f]$ in the program, $C[e]$ is valid0-stateful.
For each expression occurrence $C[e.f_{\text{state}}]$ in the program, $C[e]$ is valid0-stateful and f is declared a **rep** field.
For each expression occurrence $C[g_{\text{state}}]$ in the program, $C[g]$ is a valid0-stateful variable occurrence.
- If an expression occurrence is the target of an inspector method call or an argument for a parameter mentioned in the inspector method's **reads** clause, then it is a valid-stateful expression occurrence.

3 Program execution

The *values* are the booleans, the integers, the null reference, and the object references. Each object reference o has an associated class, denoted by $\text{classof}(o)$.

$$\mathcal{V} = \mathbb{B} \cup \mathbb{Z} \cup \{\text{null}\} \cup \mathcal{O}$$

where

$$\mathbb{B} = \{\text{true}, \text{false}\}$$

We assume as given an *operator interpretation function* interp that maps an operator to its interpretation:

$$\text{interp}(op) : \mathcal{V}^{\text{arity}(op)} \rightarrow \mathcal{V}$$

We assume all operators are total. We also assume that an operator returns only object references passed to it as an argument. Note that the nullary operators are the literals.

Note that one can see **instanceof** c , for a given c , as an operator.

Our dynamic semantics is a conservative extension of Java's semantics. It is conservative in the sense that the observable behavior of the program is the same as in Java. The only difference is that additional state is tracked and the program may get stuck where it would not get stuck in Java.

One piece of additional state is the nested object states. The heap maps object references to object states. An object state is a total map of labels to either program values or object states. (The fact that the map is total is a technical detail; an object state maps field names not declared by the object's class to null. Note that the state of an object p referenced by a field $o.f$ is typically looked up in the heap, not in the object state of o ; the state of p is looked up in the state of o only in cases where o is valid and f is a **rep** field. Since there are never cycles in reference chains formed by **rep** fields of valid objects, this is always possible.)

Definition 5. A label is either of the form f or of the form f_{state} , where f is a field name.

Definition 6. The set \mathcal{S}_n of object states of maximum depth n (with $n \in \mathbb{N}$) is the set of total functions from labels to the union of the program values and the object states of maximum depth i with $0 \leq i < n$. The set \mathcal{S} of object states is the union of \mathcal{S}_n for all $n \in \mathbb{N}$.

The set \mathcal{Z} of program variables is defined as

$$\mathcal{Z} = \mathcal{X} \cup \{\text{this}, \text{result}\}$$

The set \mathcal{R} of activation records is defined as

$$\mathcal{R} = (\mathcal{O} \leftrightarrow \mathcal{S}) \times \mathcal{P}(\mathcal{O}) \times \mathcal{O} \times \mathcal{M} \times \mathcal{V}^* \times \mathcal{X} \times (\mathcal{Z} \rightarrow \mathcal{V}) \times s^*$$

The set Σ of program states is defined as

$$\Sigma = (\mathcal{O} \hookrightarrow \mathcal{S}) \times \mathcal{P}(\mathcal{O}) \times ((\mathcal{Z} \rightarrow \mathcal{V}) \times s^*) \times \mathcal{R}^*$$

A program state $\sigma = (H, P, V, \bar{s}, F)$ consists of a heap H , a packed set P (the set of packed objects), a variable valuation V , a list of statements \bar{s} , and a call stack F , which is a list of activation records, each activation record $R = (H^{\text{pre}}, P^{\text{pre}}, o, m, \bar{v}, x, V, \bar{s})$ consisting of a pre-state heap H^{pre} , a pre-state packed set P^{pre} , a target object o , a method name m , an argument list \bar{v} , a result variable x , a variable valuation V , and a list of statements \bar{s} .

We denote the empty list by $\langle \rangle$ and the list with head h and tail t as $h \cdot t$.

The program's initial state is the state $(\emptyset, \emptyset, (\lambda g.\text{null}), \bar{s}, \langle \rangle)$ where \bar{s} is the program's main routine.

We define the relation $\text{rep}_H \subseteq \mathcal{O} \times \mathcal{O}$ as follows:

$$\text{rep}_H = \{(o, p) \bullet (\exists \text{rep } f \bullet (o, f, p) \in H)\}$$

We denote the transitive closure of a relation R by R^+ and the transitive reflexive closure by R^* . We denote the image of an element o under a relation R by $R(o)$ and the image of a set P under a relation R by $R(P)$. We call $\text{rep}_H^+(P)$ the set of *committed objects* in a state (H, P) .

3.1 Expression evaluation

Figure 2 defines a big step evaluation semantics of an expression with respect to a pair of heaps and packed sets (the old state and the current state), as well as a total mapping of variables to values and a partial mapping of variables to object states. The result of evaluation is a value and, optionally, an object state.

$H, P, H', P', V, V_S \vdash e \rightsquigarrow v, \perp$ denotes that e evaluates to value v and does not yield an object state, with respect to current state H, P and old state H', P' and variable valuation $V : \mathcal{X} \cup \{\mathbf{this}, \mathbf{result}\} \rightarrow \mathcal{V}$, a total function from variable names to values, and variable state mapping $V_S : \mathcal{X} \cup \{\mathbf{this}\} \rightarrow \mathcal{S} \cup \{\perp\}$. $H, P, H', P', V, V_S \vdash e \rightsquigarrow v, S$ denotes that e evaluates to value v and yields object state S .

The variables that carry an object state are exactly **this**, parameters of inspector methods that are in the **reads** clause, and variables bound by quantifiers over object states. The expressions that carry an object state are exactly the **rep** expressions. The value of an inspector method call depends only on the object states carried by **this** and the parameters mentioned in the **reads** clause, not on the current heap.

S denotes an object state carried by a variable or expression, whereas \tilde{S} denotes an object state or \perp .

Note that the definition of expression evaluation in this subsection and the object state validity level definitions in the next subsection are mutually recursive. Specifically, evaluation of a **forall** expression whose domain is **object.state**_{level} quantifies over **valid3**_{level}, and **valid3**_{level} is defined in terms

of expression evaluation. The recursion is well-founded since valid3_{level} is defined in terms of the evaluation of only expressions in which only domains $\text{object_state}_{level'}$ appear where $level' < level$. (This follows from program well-formedness.)

We define the following shorthand notations:

$$\begin{array}{ll}
H, P, H', P', V, V_S \vdash e \rightsquigarrow v & (\exists \tilde{S} \bullet H, P, H', P', V, V_S \vdash e \rightsquigarrow e, \tilde{S}) \\
H, P, H', P', V \vdash e \rightsquigarrow v & (\exists V_S, \tilde{S} \bullet H, P, H', P', V, V_S \vdash e \rightsquigarrow e, \tilde{S}) \\
H, P, V \vdash e \rightsquigarrow v & (\exists H', P', V_S, \tilde{S} \bullet H, P, H', P', V, V_S \vdash e \rightsquigarrow e, \tilde{S}) \\
H, P, V, V_S \vdash e \rightsquigarrow v, \tilde{S} & (\exists H', P' \bullet H, P, H', P', V, V_S \vdash e \rightsquigarrow e, \tilde{S})
\end{array}$$

Note: state-carrying variables must always hold an object (they cannot hold another value, such as a null reference). This is checked at **heap** expressions. Rep fields of packed objects must always hold an object as well. This is checked at pack operations.

Object states carried by variables or expressions are always packed, except when evaluating an object invariant.

3.2 Valid object states

We inductively define the sets $\text{valid0}_\tau, \text{valid1}_\tau, \dots$, for each class or interface τ in the program, in order. We define these multiple levels of object state validity to ensure that the recursion between the definition of object state validity and the definition of expression evaluation is well-founded. Also, the various levels are put to use when defining the verification logic and the verification conditions in Section 4. For example, the verification logic's theory includes an axiom saying that if an object state is valid2 at level c , then c 's object invariant holds for it, and if it is valid at level c , then c 's derived invariant holds for it. And importantly, the verification condition that checks the derived invariant of class c assumes that the state of **this** is valid2 , but not that it is valid .

- (o, S) is in valid0_{τ_i} if for each substate (o', S') of (o, S) , either the class c of o' is not declared by the program or for each rep field f declared by c , $S'(f)$ is an object reference. $\text{valid0}_{\tau_{i+1}}$ is equal to valid_{τ_i} .
- If for all object states (o, S) in valid0_{τ_i} where the class of o is τ , evaluation of the invariant declared by τ yields a boolean, then valid1_{τ_i} is valid0_{τ_i} ; otherwise it is the empty set.
- (o, S) is in valid2_{τ_i} iff it is in valid1_{τ_i} and for each substate (o', S') , either τ is an interface or the class of o' is not τ or evaluation of the invariant declared by τ on (o', S') yields true.
- If, for each call of inspector method r_j declared by τ_i , where all argument states are in $\text{valid3}_{\tau_i, r_{j-1}}$, evaluation of the precondition yields a boolean, and if it yields true, evaluation of the body does not get stuck, then $\text{valid3}_{\tau_i, r_j}$ equals $\text{valid3}_{\tau_i, r_{j-1}}$; otherwise, it equals the empty set.

$$\begin{array}{c}
H, P, H', P', V, V_S \vdash g \rightsquigarrow V(g), \perp \qquad H, P, H', P', V, V_S \vdash g_{\text{state}} \rightsquigarrow V(g), V_S(g) \\
\\
\frac{H, P, H', P', V, V_S \vdash e_1, \dots, e_n \rightsquigarrow v_1, \tilde{S}_1, \dots, v_n, \tilde{S}_n}{H, P, H', P', V, V_S \vdash \text{op}(e_1, \dots, e_n) \rightsquigarrow \text{interp}(\text{op})(v_1, \dots, v_n), \perp} \\
\\
\frac{H, P, H', P', V, V_S \vdash e \rightsquigarrow o \quad \text{classof}(o) = \text{declaringClass}(f)}{H, P, H', P', V, V_S \vdash e.f_{\text{heap}} \rightsquigarrow H(o)(f), \perp} \\
\\
\frac{H, P, H', P', V, V_S \vdash e \rightsquigarrow o, S \quad \text{classof}(o) = \text{declaringClass}(f)}{H, P, H', P', V, V_S \vdash e.f \rightsquigarrow S(f), \perp} \\
\\
\frac{H, P, H', P', V, V_S \vdash e \rightsquigarrow o, S \quad \text{classof}(o) = \text{declaringClass}(f)}{H, P, H', P', V, V_S \vdash e.f_{\text{state}} \rightsquigarrow S(f), S(f_{\text{state}})} \\
\\
\frac{H, P, H', P', V, V_S \vdash e \rightsquigarrow o \quad o \in P}{H, P, H', P', V, V_S \vdash \text{heap}(e) \rightsquigarrow o, H(o)} \qquad \frac{H, P, H', P', V, V_S \vdash e \rightsquigarrow o}{H, P, H', P', V, V_S \vdash e.\text{inv} \rightsquigarrow (o \in P), \perp} \\
\\
\frac{H, P, H', P', V, V_S \vdash e \rightsquigarrow o}{H, P, H', P', V, V_S \vdash e.\text{committed} \rightsquigarrow (o \in \text{rep}_H^+(P)), \perp} \\
\\
\frac{\begin{array}{l} H, P, H', P', V, V_S \vdash e, e_1, \dots, e_n \rightsquigarrow o, S, v_1, \tilde{S}_1, \dots, v_n, \tilde{S}_n \\ (\forall i \in \{1, \dots, n\}) \bullet x_i \in \{\bar{x}\} \Rightarrow S_i = \tilde{S}_i \quad \text{classof}(o) = c \quad c \preceq \tau \\ \text{class } c \dots \{ \dots \text{inspector } r(x_1, \dots, x_n) \text{ reads } \bar{x}; \text{ requires } e_P; \{ \text{return } e_{\text{body}}; \} \dots \} \\ H, P, H', P', (\lambda g.\text{null})[\text{this} \mapsto o, \bar{x} \mapsto \bar{v}], (\lambda g.\perp)[\text{this} \mapsto S, \bar{x} \mapsto \bar{S}] \vdash e_P \rightsquigarrow \text{true} \\ H, P, H', P', (\lambda g.\text{null})[\text{this} \mapsto o, \bar{x} \mapsto \bar{v}], (\lambda g.\perp)[\text{this} \mapsto S, \bar{x} \mapsto \bar{S}] \vdash e_{\text{body}} \rightsquigarrow v \end{array}}{H, P, H', P', V, V_S \vdash e.r_\tau(e_1, \dots, e_n) \rightsquigarrow v, \perp} \\
\\
\frac{H', P', H', P', V, V_S \vdash e \rightsquigarrow v, \tilde{S}}{H, P, H', P', V, V_S \vdash \text{old}(e) \rightsquigarrow v, \tilde{S}} \\
\\
\frac{H, P, H', P', V, V_S \vdash e_1 \rightsquigarrow \text{true} \quad H, P, H', P', V, V_S \vdash e_2 \rightsquigarrow v, \tilde{S}}{H, P, H', P', V, V_S \vdash e_1 ? e_2 : e_3 \rightsquigarrow v, \tilde{S}} \\
\\
\frac{H, P, H', P', V, V_S \vdash e_1 \rightsquigarrow \text{false} \quad H, P, H', P', V, V_S \vdash e_3 \rightsquigarrow v, \tilde{S}}{H, P, H', P', V, V_S \vdash e_1 ? e_2 : e_3 \rightsquigarrow v, \tilde{S}} \\
\\
\frac{\begin{array}{l} H, P, H', P', V, V_S \vdash e_1, e_2 \rightsquigarrow n_1, n_2 \\ \forall n \bullet n_1 \leq n < n_2 \Rightarrow H, P, H', P', V[x \mapsto n], V_S \vdash e_3 \rightsquigarrow b_n \end{array}}{H, P, H', P', V, V_S \vdash \text{forall}\{x \text{ in } (e_1 : e_2); e_3\} \rightsquigarrow (\forall n \bullet n_1 \leq n < n_2 \Rightarrow b_n), \perp} \\
\\
\frac{\forall o \bullet H, P, H', P', V[x \mapsto o], V_S \vdash e \rightsquigarrow b_o}{H, P, H', P', V, V_S \vdash \text{forall}\{x \text{ in object}; e\} \rightsquigarrow (\forall o \bullet b_o), \perp} \\
\\
\frac{\forall (o, S) \in \mathfrak{D} \bullet H, P, H', P', V[x \mapsto o], V_S[x \mapsto S] \vdash e \rightsquigarrow b_{o,S}}{H, P, H', P', V, V_S \vdash \text{forall}\{x \text{ in object.state}_{\text{level}}; e\} \rightsquigarrow (\forall (o, S) \in \text{valid3}_{\text{level}} \bullet b_{o,S}), \perp}
\end{array}$$

Figure 2: Expression evaluation

- We distinguish two cases:
 - Assume τ is an interface. If, for each element (o, S) of $\text{valid3}_{\tau_i, r_m}$, either the class of o does not implement τ_i or evaluation of the dynamic invariant declared by τ on (o, S) , with quantification over $\text{valid3}_{\tau_i, r_m}$, does not get stuck and yields **true**, then valid_{τ_i} equals $\text{valid3}_{\tau_i, r_m}$; otherwise, it equals the empty set.
 - Assume τ is a class. If, for each element (o, S) of $\text{valid3}_{\tau_i, r_m}$, either the class of o is not τ_i or evaluation of the derived invariant declared by τ on (o, S) , with quantification over $\text{valid3}_{\tau_i, r_m}$, does not get stuck and yields **true**, then valid_{τ_i} equals $\text{valid3}_{\tau_i, r_m}$; otherwise, it equals the empty set.

We define valid3_τ to be equal to $\text{valid3}_{\tau, r_m}$. We define **valid** to be equal to valid_{τ_n} . We define valid_∞ to be equal to **valid**.

3.3 Statement execution

Figure 3 gives a small step semantics. An execution step is denoted as $\rightarrow \subseteq \Sigma \times \Sigma$.

4 Program validity

In this section, we define program validity. In the next section, we prove that valid programs do not get stuck. A program is valid if each method is valid.

We introduce a third state variable, namely the set C of committed objects. We maintain the invariant that

$$C = \text{rep}_H^+(P)$$

4.1 Verification logic

4.1.1 Syntax

We use a classical multi-sorted first-order predicate logic. That is, a *term* t is a *logical variable* $y \in Y$ or a *function application* $f(t_1, \dots, t_n)$ where f is a function symbol of the signature with arity n . A formula ϕ is an *equality* $t_1 = t_2$, a *propositional formula* using the connectives \wedge , \vee , \Rightarrow , and \neg , or a quantification $(\forall y \bullet \phi)$.

4.1.2 Signature

The *base sorts* are the following:

- The sort of booleans
- The sort of integers

$$\begin{array}{c}
H, P, V \vdash e \rightsquigarrow v \\
\text{class } c \cdots \{ \cdots \text{ mutator } m(\bar{p}) \text{ requires } e_P; \text{ modifies } \bar{M}; \text{ ensures } e_Q; \{ \bar{s}' \} \cdots \} \\
\frac{
\begin{array}{c}
H, P, H^{\text{old}}, P^{\text{old}}, (\lambda g.\text{null})[\text{this} \mapsto o, \bar{p} \mapsto \bar{v}, \text{result} \mapsto v] \vdash e_Q \rightsquigarrow \text{true} \\
H^{\text{old}}, P^{\text{old}}, (\lambda g.\text{null})[\text{this} \mapsto o, \bar{p} \mapsto \bar{v}] \vdash \bar{M} \mapsto \bar{o} \\
(\forall p \in \text{dom}(H^{\text{old}}) \bullet p \notin \text{rep}_{H^{\text{old}}}^+(P^{\text{old}}) \wedge p \notin \{ \bar{o} \} \Rightarrow \\
H(p) = H^{\text{old}}(p) \wedge p \notin \text{rep}_{H^{\text{old}}}^+(P^{\text{old}}) \wedge (p \in P) = (p \in P^{\text{old}}))
\end{array}
}{
(H, P, V, \text{return } e; , (H^{\text{old}}, P^{\text{old}}, o, m, \bar{v}, x, V', \bar{s}) \cdot F) \rightarrow (H, P, V'[x \mapsto v], \bar{s}, F)
} \\
\\
\frac{
H, P, V \vdash e \rightsquigarrow v
}{
(H, P, V, x := e; \bar{s}, F) \rightarrow (H, P, V[x \mapsto v], \bar{s}, F)
} \\
\\
\frac{
V(g_1) = o \quad \text{classof}(o) = c \quad \text{declaringclass}(f) = c \quad o \notin P
}{
(H, P, V, g_1.f := g_2; ss, F) \rightarrow (H[(o, f) \mapsto V(g_2)], P, V, \bar{s}, F)
} \\
\\
\frac{
\begin{array}{c}
V(g_t) = o \\
\text{classof}(o) = c \quad \text{class } c \cdots \{ \cdots m(\bar{p}) \text{ requires } e_P; \text{ modifies } \bar{M}; \text{ ensures } e_Q; \{ \bar{s}' \} \\
H, P, (\lambda g.\text{null})[\text{this} \mapsto o, \bar{p} \mapsto V(\bar{g})] \vdash e_P \rightsquigarrow \text{true}
\end{array}
}{
(H, P, V, \text{call } x := g_t.m(\bar{g}); \bar{s}, F)
} \\
\downarrow \\
(H, P, \lambda g.\text{null})[\text{this} \mapsto o, \bar{p} \mapsto V(\bar{g})], \bar{s}', (H, P, o, m, V(\bar{g}), x, V, \bar{s}, F) \\
\\
\frac{
o \notin \text{dom}(H) \quad \text{classof}(o) = c \quad \text{fields}(c) = \bar{f}
}{
(H, P, V, x := \text{new } c; \bar{s}, F) \rightarrow (H[o \mapsto (\lambda \ell.\text{null})[\bar{f} \mapsto \text{null}]], P, V[x \mapsto o], \bar{s}, F)
} \\
\\
\frac{
\begin{array}{c}
V(g) = o \quad \text{classof}(o) = c \\
o \notin P \quad \text{rep}_H(o) \subseteq P \setminus \text{rep}_H^+(P) \quad \text{class } c \cdots \{ \cdots \text{invariant } I; \cdots \} \\
S' = H(o)[(f_1)_{\text{state}} \mapsto H(H(o)(f_1))] \cdots [(f_n)_{\text{state}} \mapsto H(H(o)(f_n))] \\
H, P, (\lambda g.\text{null})[\text{this} \mapsto o], (\lambda g.\perp)[\text{this} \mapsto S'] \vdash I \rightsquigarrow \text{true}
\end{array}
}{
(H, P, V, \text{pack}_c g; \bar{s}, F)
} \\
\downarrow \\
(H[o \mapsto S'], P \cup \{o\}, V, \bar{s}) \\
\\
\frac{
V(g) = o \quad \text{classof}(o) = c \quad o \in P \setminus \text{rep}_H^+(P)
}{
(H, P, V, \text{unpack}_c g; \bar{s}, F) \rightarrow (H, P \setminus \{o\}, V, \bar{s})
} \\
\\
\frac{
H, P, V \vdash e \rightsquigarrow \text{true}
}{
(H, P, V, \text{if } (e) \{ \bar{s}_1 \} \text{ else } \{ \bar{s}_2 \} \bar{s}, F) \rightarrow (H, P, V, \bar{s}_1 \bar{s}, F)
} \\
\\
\frac{
H, P, V \vdash e \rightsquigarrow \text{false}
}{
(H, P, V, \text{if } (e) \{ \bar{s}_1 \} \text{ else } \{ \bar{s}_2 \} \bar{s}, F) \rightarrow (H, P, V, \bar{s}_2 \bar{s}, F)
}
\end{array}$$

Figure 3: Statement execution

- The sort of object references
- The sort of program values
- The sort of field names
- The sort of class names
- The sort of interface names
- The sort of object states
- The sort of labels

The sorts are defined inductively as follows:

- Each base sort is a sort
- If s_1 and s_2 are sorts, then the following are sorts as well:
 - the powerset sort $\mathcal{P}(s_1)$
 - the partial function sort $s_1 \leftrightarrow s_2$

The signature consists of:

- Function symbols for dealing with sets: `emptyset`, `insert`, `contains`, `union`, `diff`.
- Function symbols for dealing with partial functions: `dom`, `apply`, `update`.
- A symbol for the null value: `null`. (This symbol belongs to the sort of the program values, not the object references. The null value is not an object reference.)
- Function symbols that tell whether a program value is of a particular type: `isbool`, `isint`, `isobj`.
- Function symbols for casting program values: `val2bool`, `bool2val`, `val2int`, `int2val`, `val2ref`, `ref2val`.
- For each class name c , a function symbol c
- For each interface name I , a function symbol I
- A function symbol that denotes the class of an object reference: `classof`.
- For each field name f , a function symbol f
- For each operator op , a function symbol `optermop`
- Function symbols for dealing with object states: `os_apply`, `os_update`.

- For each inspector method r , a function symbol insfunc_r . If r has n parameters and mentions m parameters in its **reads** clause, then the arity of insfunc_r is $2 + m + n$.
- Binary function symbols valid0_τ , valid1_τ , valid2_τ , $\text{valid3}_{\tau,r}$, and valid_τ , for each class or interface τ declared by the program, and for each inspector method r declared by τ , that take an object reference and an object state and return a boolean, and a ternary function symbol reachable that takes a heap, a packed set, and a committed set and returns a boolean.

4.1.3 Interpretation

Throughout, we interpret the logic using a fixed interpretation \mathcal{J} , which is as expected. If an argument is outside of the natural domain of a function, the function returns an arbitrary but fixed value of the range sort (fixed in the sense that for a given argument list, the interpretation of a given function symbol yields a fixed result value). For example, $\mathcal{J}(\text{val2ref})(\text{null})$ returns some object reference.

For an inspector method

inspector $r(\bar{p})$ **reads** \bar{x} ; **requires** e_P ; { **return** e_{body} ; }

the interpretation $\mathcal{J}(\text{insfunc}_r)(S, S_1, \dots, S_m, o, v_1, \dots, v_n)$ is the unique value v such that

$$\emptyset, \emptyset, (\lambda g.\text{null})[\mathbf{this} \mapsto o, \bar{p} \mapsto \bar{v}], (\lambda g.\perp)[\mathbf{this} \mapsto S, \bar{x} \mapsto \bar{S}] \vdash e_{\text{body}} \rightsquigarrow v$$

or null if no such value exists.

$\mathcal{J}(\text{reachable})(H, P, C)$ is true if and only if for each object reference o in the domain of H , either the class c of o is not declared by the program or both 1) for each field f declared by c , if $H(o, f)$ is an object reference then $H(o, f) \in \text{dom}(H)$, and 2) if $o \in P$ then $H(o)$ is a valid object state for o (i.e., $(o, H(o))$ is in valid).

4.1.4 Theory

Let Σ be some finite axiomatization of \mathcal{J} . We use Σ to attempt to prove the verification conditions. Note: we do not require Σ to be complete since no complete finite (or even computable) axiomatization of our interpretation exists (since it includes the natural numbers).

For each class

class c **implements** \bar{I} { ... **invariant** e_I ; ... **derived_invariant** e_d ; ... }

the theory includes an axiom

$$\forall o, S \bullet \text{valid2}_c(o, S) \Rightarrow \text{classof}(o) = c \Rightarrow \text{ev}(e_I)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}]$$

and an axiom

$$\forall o, S \bullet \text{valid}_c(o, S) \Rightarrow \text{classof}(o) = c \Rightarrow \text{ev}(e_d)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}]$$

For each interface

$$\mathbf{interface} \ I \ \{ \dots \ \mathbf{dynamic_invariant} \ e_D; \dots \}$$

the theory includes an axiom

$$\forall o, S \bullet \text{valid}_I(o, S) \Rightarrow \text{classof}(o) \prec I \Rightarrow \text{ev}(e_D)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}]$$

We axiomatize the inspector method functions as follows.

For each inspector method

$$\mathbf{inspector} \ r_j(x_1, \dots, x_n) \ \mathbf{reads} \ x_{k_1}, \dots, x_{k_m}; \ \mathbf{requires} \ e_P; \ \{ \mathbf{return} \ e; \}$$

declared by class c the theory contains the axiom

$$\begin{aligned} & (\forall S, S_1, \dots, S_m, o, v_1, \dots, v_n \bullet \\ & \quad \text{classof}(o) = c \wedge \text{valid}_{3_{c,r_j}}(o, S) \Rightarrow \\ & \quad \text{isobj}(v_{k_1}) \wedge \text{valid}_{3_{c,r_j}}(v_{k_1}, S_1) \wedge \dots \wedge \text{isobj}(v_{k_m}) \wedge \text{valid}_{3_{c,r_j}}(v_{k_m}, S_m) \Rightarrow \\ & \quad \text{ev}(e_P) \Rightarrow \\ & \quad \text{insfunc}_r(S, S_1, \dots, S_m, o, v_1, \dots, v_n) = \\ & \quad \text{ev}(e)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}, \bar{v}/\bar{p}, \bar{S}/\bar{x}_{\text{state}}]) \end{aligned}$$

Note that if an inspector method is declared by an interface and implemented by more than one class, then the theory contains multiple axioms for the same function symbol; however, the axioms are consistent since they have mutually exclusive premises, in particular the premise saying that the class of the receiver object equals the inspector method's declaring class.

For each two consecutive validity levels $\text{valid}A$ and $\text{valid}B$, we have the axiom:

$$\forall o, S \bullet \text{valid}B(o, S) \Rightarrow \text{valid}A(o, S)$$

The theory includes the following axiom:

$$\forall H, P, C \bullet \text{reachable}(H, P, C) \Rightarrow (\forall o \bullet o \in P \Rightarrow \text{valid}(H(o)))$$

4.2 Expression validity and value

We use $\text{ve}(e)$ to denote the expression validity verification condition of expression e (i.e., the logical formula that denotes the validity of e), $\text{ev}(e)$ to denote the logical term that denotes the value of e , and $\text{es}(e)$ to denote the logical term that denotes the object state carried by e . Their definitions are given in Figure 4. $\text{es}(e)$ is defined only for stateful expressions e .

The free variables of these formulae and expressions are included in

$$\{H, P, C, H^{\text{old}}, P^{\text{old}}, C^{\text{old}}\} \cup \{g, g_{\text{state}} \mid g \in \mathcal{X} \cup \{\mathbf{this}, \mathbf{result}\}\}$$

$$\begin{aligned}
\text{ve}(g) &\equiv \mathbf{true} & \text{ev}(g) &\equiv g \\
\text{ve}(g_{\text{state}}) &\equiv \mathbf{true} & \text{ev}(g_{\text{state}}) &\equiv g & \text{es}(g_{\text{state}}) &\equiv g_{\text{state}} \\
\text{ve}(op(e_1, \dots, e_n)) &\equiv \text{ve}(e_1) \wedge \dots \wedge \text{ve}(e_n) \\
\text{ev}(op(e_1, \dots, e_n)) &\equiv \text{opterm}(op, \text{ev}(e_1), \dots, \text{ev}(e_n)) \\
\text{ve}(e.f_{\text{heap}}) &\equiv \text{ve}(e) \wedge \text{isobj}(\text{ev}(e)) \wedge \text{classof}(\text{ev}(e)) = \text{declaringClass}(f) \\
\text{ev}(e.f_{\text{heap}}) &\equiv H(\text{ve}(e))(f) \\
\text{ve}(e.f) &\equiv \text{ve}(e) \wedge \text{classof}(\text{ev}(e)) = \text{declaringClass}(f) \\
\text{ev}(e.f) &\equiv \text{es}(e)(f) \\
\text{ve}(e.f_{\text{state}}) &\equiv \text{ve}(e) \wedge \text{classof}(\text{ev}(e)) = \text{declaringClass}(f) \\
\text{ev}(e.f_{\text{state}}) &\equiv \text{es}(e)(f) \\
\text{es}(e.f_{\text{state}}) &\equiv \text{es}(e)(f_{\text{state}}) \\
\text{ve}(\mathbf{heap}(e)) &\equiv \text{ve}(e) \wedge \text{isobj}(\text{ev}(e)) \wedge \text{ev}(e) \in P \\
\text{ev}(\mathbf{heap}(e)) &\equiv \text{ev}(e) \\
\text{es}(\mathbf{heap}(e)) &\equiv H(\text{ev}(e)) \\
\text{ve}(e.\mathbf{inv}) &\equiv \text{ve}(e) \wedge \text{isobj}(\text{ev}(e)) \\
\text{ev}(e.\mathbf{inv}) &\equiv (\text{ev}(e) \in P) \\
\text{ve}(e.\mathbf{committed}) &\equiv \text{ve}(e) \wedge \text{isobj}(\text{ev}(e)) \\
\text{ev}(e.\mathbf{committed}) &\equiv (\text{ev}(e) \in C) \\
\text{ve}(e.r_\tau(e_1, \dots, e_n)) &\equiv \\
&\quad \text{ve}(e) \wedge \text{ve}(e_1) \wedge \dots \wedge \text{ve}(e_n) \wedge \text{classof}(\text{ev}(e)) \preceq \tau \\
&\quad \wedge \text{ev}(e_P) [\\
&\quad \quad \text{ev}(e) / \mathbf{this}, \text{ev}(\bar{e}) / \bar{p}, \text{es}(e) / \mathbf{this}_{\text{state}}, \text{es}(e_{k_1}, \dots, e_{k_m}) / (p_{k_1}, \dots, p_{k_m})]] \\
\text{ev}(e.r_\tau(e_1, \dots, e_n)) &\equiv \\
&\quad \text{inspfunc}_\tau(\text{es}(e), \text{es}(e_{k_1}), \dots, \text{es}(e_{k_m}), \text{ev}(e), \text{ev}(e_1), \dots, \text{ev}(e_n)) \\
&\quad \text{where type } \tau \text{ declares } \mathbf{inspector } r(x_1, \dots, x_n) \mathbf{reads } x_{k_1}, \dots, x_{k_m}; \dots \\
\text{ve}(\mathbf{old}(e)) &\equiv \text{ve}(e) [H^{\text{old}} / H, P^{\text{old}} / P, C^{\text{old}} / C] \\
\text{ev}(\mathbf{old}(e)) &\equiv \text{ev}(e) [H^{\text{old}} / H, P^{\text{old}} / P, C^{\text{old}} / C] \\
\text{ve}(\mathbf{forall}\{x \text{ in } (e_1 : e_2); e_3\}) &\equiv \\
&\quad \text{ve}(e_1) \wedge \text{ve}(e_2) \\
&\quad \wedge (\forall n \bullet \text{ev}(e_1) \leq n \wedge n < \text{ev}(e_2) \Rightarrow \text{ve}(e_3)[n/x]) \\
\text{ev}(\mathbf{forall}\{x \text{ in } (e_1 : e_2); e_3\}) &\equiv \\
&\quad (\forall n \bullet \text{ev}(e_1) \leq n \wedge n < \text{ev}(e_2) \Rightarrow \text{ev}(e_3)[n/x]) \\
\text{ve}(\mathbf{forall}\{x \text{ in object}; e\}) &\equiv (\forall o \bullet \text{ve}(e)[o/x]) \\
\text{ev}(\mathbf{forall}\{x \text{ in object}; e\}) &\equiv (\forall o \bullet \text{ev}(e)[o/x]) \\
\text{ve}(\mathbf{forall}\{x \text{ in object.state}_{\text{level}}; e\}) &\equiv \\
&\quad (\forall o, S \bullet \text{valid3}_{\text{level}}(o, S) \Rightarrow \text{ve}(e)[o/x, S/x_{\text{state}}]) \\
\text{ev}(\mathbf{forall}\{x \text{ in object.state}_{\text{level}}; e\}) &\equiv \\
&\quad (\forall o, S \bullet \text{valid3}_{\text{level}}(o, S) \Rightarrow \text{ev}(e)[o/x, S/x_{\text{state}}]) \\
\text{ve}(e_1 ? e_2 : e_3) &\equiv \text{ve}(e_1) \wedge (\text{if } \text{ev}(v_1) \text{ then } \text{ve}(v_2) \text{ else } \text{ve}(v_3)) \\
\text{ev}(e_1 ? e_2 : e_3) &\equiv (\text{if } \text{ev}(v_1) \text{ then } \text{ev}(v_2) \text{ else } \text{ev}(v_3)) \\
\text{es}(e_1 ? e_2 : e_3) &\equiv (\text{if } \text{ev}(v_1) \text{ then } \text{es}(v_2) \text{ else } \text{es}(v_3))
\end{aligned}$$

Figure 4: Expression validity, expression value, and expression state

4.3 Continuation validity

We use $\text{vs}(\bar{s}, Q)$ to denote the continuation verification condition of continuation \bar{s} with respect to postcondition Q . It is defined in Figure 5.

The free variables of these formulae are included in

$$\{H, P, C, H^{\text{old}}, P^{\text{old}}, C^{\text{old}}\} \cup \{g, g_{\text{state}} \mid g \in \mathcal{X} \cup \{\mathbf{this}\}\}$$

4.4 Program validity

Definition 7. The closure $\text{close}(\phi)$ of a verification condition ϕ is ϕ after substitution of null for all logical variables of the form g , where g is a program variable, that appear free in ϕ .

Definition 8. An invariant e declared by a class c is valid if it is a valid expression under the condition that the rep fields of \mathbf{this} hold object references and their states are valid.

$$\begin{array}{c} \Sigma \vdash \\ \forall o, S \bullet \\ \text{classof}(o) = c \wedge \text{valid0}_c(o, S) \Rightarrow \\ \text{ve}(e)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}] \end{array}$$

Definition 9. An inspector method header

$$\mathbf{inspector} \ r_j(\bar{p}) \ \mathbf{reads} \ \bar{x}; \ \mathbf{requires} \ e_P;$$

declared by a type τ is valid if the precondition is a valid expression:

$$\begin{array}{c} \Sigma \vdash \\ \forall o, S, \bar{v}, \bar{S} \bullet \\ \text{classof}(o) \preceq \tau \wedge \text{valid3}_{\tau, r_{j-1}}(o, S) \\ \wedge \text{isobj}(v_{k_1}) \wedge \text{valid3}_{\tau, r_{j-1}}(v_{k_1}, S_1) \wedge \cdots \wedge \text{isobj}(v_{k_m}) \wedge \text{valid3}_{\tau, r_{j-1}}(v_{k_m}, S_m) \\ \Downarrow \\ \text{close}(\text{ve}(e_P)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}, \bar{v}/\bar{p}, \bar{S}/\bar{x}_{\text{state}}]) \end{array}$$

Definition 10. The body of an inspector method

$$\mathbf{inspector} \ r(\bar{p}) \ \mathbf{reads} \ \bar{x}; \ \mathbf{requires} \ e_P; \ \{ \ \mathbf{return} \ e; \ \}$$

declared by a class c is valid if the body is a valid expression:

$$\begin{array}{c} \Sigma \vdash \\ \forall o, S, \bar{v}, \bar{S} \bullet \\ \text{classof}(o) = c \wedge \text{valid3}_{\tau, r_{j-1}}(o, S) \\ \wedge \text{isobj}(v_{k_1}) \wedge \text{valid3}_{\tau, r_{j-1}}(v_{k_1}, S_1) \wedge \cdots \wedge \text{isobj}(v_{k_m}) \wedge \text{valid3}_{\tau, r_{j-1}}(v_{k_m}, S_m) \\ \Downarrow \\ \text{close}(\text{ev}(e_P)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}, \bar{v}/\bar{p}, \bar{S}/\bar{x}_{\text{state}}]) \\ \Downarrow \\ \text{close}(\text{ve}(e)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}, \bar{v}/\bar{p}, \bar{S}/\bar{x}_{\text{state}}]) \end{array}$$

$$\begin{aligned}
\text{vs}(\mathbf{return } e; , Q) &\equiv \text{ve}(e) \wedge Q[\text{ev}(e)/\mathbf{result}] \\
\text{vs}(x := e; \bar{s}, Q) &\equiv \text{ve}(e) \wedge \text{vs}(\bar{s}, Q)[\text{ev}(e)/x] \\
\text{vs}(g_1.f := g_2; \bar{s}, Q) &\equiv \\
&g_1 \neq \text{null} \wedge \text{classof}(g_1) = \text{declaringClass}(f) \wedge g_1 \notin P \\
&\wedge \text{vs}(\bar{s}, Q)[H[(g_1, f) \mapsto g_2]/H] \\
\text{vs}(\mathbf{call } x := g_t.m(\bar{g}); \bar{s}, Q) &\equiv \\
&g_t \neq \text{null} \wedge \text{classof}(g_t) \preceq \text{declaringType}(m) \wedge \text{ev}(e_P)[g_t/\mathbf{this}, \bar{g}/\bar{p}] \\
&\wedge \text{vcc}(\mathbf{call } x := g_t.m(\bar{g}); \bar{s}, Q) \\
&\text{where } \pi = \dots m(\bar{p}) \mathbf{requires } e_P; \mathbf{modifies } \bar{M}; \mathbf{ensures } e_Q; \dots \\
\text{vcc}(\mathbf{call } x := g_t.m(\bar{g}); \bar{s}, Q) &\equiv \\
&(\forall H', P', C', v_r \bullet \\
&\quad (\text{dom}(H) \subseteq \text{dom}(H') \wedge v_r \in \text{dom}(H') \\
&\quad \wedge \text{ev}(e_Q)[g_t/\mathbf{this}, \bar{g}/\bar{p}, v_r/\mathbf{result}, \\
&\quad \quad H'/H, P'/P, C'/C, H/H^{\text{old}}, P/P^{\text{old}}, C/C^{\text{old}}] \\
&\quad \wedge (\forall o \bullet o \in \text{dom}(H) \wedge o \notin C^{\text{pre}} \wedge o \notin \{\text{ev}(\bar{M})[g_t/\mathbf{this}, \bar{g}/\bar{p}, H'/H]\} \\
&\quad \quad \Rightarrow (H'(o) = H(o) \wedge (o \in P') = (o \in P) \wedge (o \in C') = (o \in C))) \\
&\quad \Rightarrow (\text{reachable}(H, P, C) \Rightarrow \text{vs}(\bar{s}, Q))[v_r/x, H'/H, P'/P, C'/C]) \\
&\quad \text{where } \pi = \dots m(\bar{p}) \mathbf{requires } e_P; \mathbf{modifies } \bar{M}; \mathbf{ensures } e_Q; \dots \\
\text{vs}(x := \mathbf{new } c; \bar{s}, Q) &\equiv \\
&(\forall o \bullet \text{classof}(o) = c \wedge o \notin \text{dom}(H) \wedge o \notin P \wedge o \notin C \\
&\Rightarrow \text{vs}(\bar{s}, Q)[o/x, H[o \mapsto \emptyset[\bar{f} \mapsto \text{null}]]/H]) \\
&\text{where } \text{fields}(c) = \bar{f} \\
\text{vs}(\mathbf{pack}_c g; \bar{s}, Q) &\equiv \\
&\text{isobj}(g) \wedge \text{classof}(g) = c \wedge g \notin P \\
&\wedge \text{isobj}(H(g, f_1)) \wedge H(g, f_1) \in P \setminus C \wedge \dots \wedge \text{isobj}(H(g, f_n)) \\
&\wedge H(g, f_n) \in P \setminus C \wedge \text{ev}(e)[g/\mathbf{this}, H(g)/\mathbf{this}_{\text{state}}] \\
&\wedge (\text{reachable}(H, P, C) \Rightarrow \\
&\quad \text{vs}(\bar{s}, Q)[(P \cup \{x\})/P, (C \cup \{H(g, f_1), \dots, H(g, f_n)\})/C]) \\
&\text{where } \pi = \dots \mathbf{class } c \dots \{ \dots \mathbf{invariant } e; \dots \} \dots \\
&\quad \text{and } f_1, \dots, f_n \text{ are the } \mathbf{rep} \text{ fields of class } c \\
\text{vs}(\mathbf{unpack}_c g; \bar{s}, Q) &\equiv \\
&\text{isobj}(g) \wedge \text{classof}(g) = c \wedge g \in P \setminus C \\
&\wedge \text{vs}(\bar{s}, Q)[(P \setminus \{g\})/P, (C \setminus \{H(g, f_1), \dots, H(g, f_n)\})/C] \\
&\text{where } f_1, \dots, f_n \text{ are the } \mathbf{rep} \text{ fields of class } c \\
\text{vs}(\mathbf{if } (e) \{ \bar{s}_1 \} \mathbf{else } \{ \bar{s}_2 \} \bar{s}, Q) &\equiv \\
&\text{ve}(e) \wedge (\text{ev}(e) \Rightarrow \text{vs}(\bar{s}_1 \bar{s}, Q)) \wedge (\neg \text{ev}(e) \Rightarrow \text{vs}(\bar{s}_2 \bar{s}, Q))
\end{aligned}$$

Figure 5: Statement validity

Definition 11. A derived invariant e declared by a class c is valid if it is a valid expression and it is true under the condition that the receiver is valid.

$$\begin{array}{c} \Sigma \vdash \\ \forall o, S \bullet \\ \text{classof}(o) = c \Rightarrow \\ \text{ve}(e)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}] \wedge (\text{valid}_{3_{c,r_m}}(o, S) \Rightarrow \text{ev}(e)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}]) \end{array}$$

Definition 12. A dynamic invariant e declared by an interface I is valid if it is a valid expression under the condition that the receiver is valid.

$$\begin{array}{c} \Sigma \vdash \\ \forall o, S \bullet \\ \text{classof}(o) \prec I \Rightarrow \\ \text{ve}(e)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}] \end{array}$$

Definition 13. A class c correctly implements an interface I if the dynamic invariant e declared by I holds for valid objects of class c .

$$\begin{array}{c} \Sigma \vdash \\ \forall o, S \bullet \\ \text{classof}(o) = c \wedge \text{valid}_{3_{c,r_m}}(o, S) \Rightarrow \\ \text{ev}(e)[o/\mathbf{this}, S/\mathbf{this}_{\text{state}}] \end{array}$$

Definition 14. A mutator method header

mutator $m(\bar{p})$ **requires** e_P ; **modifies** $e_1^{\text{mod}}, \dots, e_n^{\text{mod}}$; **ensures** e_Q ;

declared by a type τ is valid if the precondition, the modifies clause items, and the postcondition are valid expressions.

$$\begin{array}{c} \Sigma \vdash \\ \forall H^{\text{old}}, P^{\text{old}}, C^{\text{old}}, H, P, C, o, \bar{v}, v_r \bullet \\ \text{classof}(o) = c \wedge \text{reachable}(H^{\text{old}}, P^{\text{old}}, C^{\text{old}}) \wedge \text{reachable}(H, P, C) \\ \downarrow \\ \text{close}(\text{ve}(e_P)[o/\mathbf{this}, \bar{v}/\bar{p}, H^{\text{old}}/H, P^{\text{old}}/P, C^{\text{old}}/C]) \\ \wedge \\ (\text{close}(\text{ev}(e_P)[o/\mathbf{this}, \bar{v}/\bar{p}, H^{\text{old}}/H, P^{\text{old}}/P, C^{\text{old}}/C]) \\ \downarrow \\ \text{close}(\text{ve}(e_1^{\text{mod}})[o/\mathbf{this}, \bar{v}/\bar{p}, H^{\text{old}}/H, P^{\text{old}}/P, C^{\text{old}}/C]) \\ \wedge \\ \vdots \\ \wedge \\ \text{close}(\text{ve}(e_n^{\text{mod}})[o/\mathbf{this}, \bar{v}/\bar{p}, H^{\text{old}}/H, P^{\text{old}}/P, C^{\text{old}}/C]) \\ \wedge \\ \text{close}(\text{ve}(e_Q)[o/\mathbf{this}, \bar{v}/\bar{p}, v_r/\mathbf{result}])) \end{array}$$

Definition 15. The effective postcondition $\text{post}(o.m(\bar{v}))$ of a call $o.m(\bar{v})$ of a mutator method with header

mutator $m(\bar{p})$ **requires** e_P ; **modifies** $e_1^{\text{mod}}, \dots, e_n^{\text{mod}}$; **ensures** e_Q ;

is given by:

$$\begin{aligned}
& \text{post}(o.m(\bar{v})) \equiv \\
& \text{close}(\text{ev}(e_Q)[o/\mathbf{this}, \bar{v}/\bar{p}, \text{result}/\mathbf{result}]) \\
& \quad \wedge \\
& \quad (\forall p \in \text{dom}(H^{\text{old}}) \bullet \\
& \quad \quad p \notin C^{\text{old}} \\
& \quad \quad \wedge \\
& \quad \quad p \neq \text{close}(\text{ev}(e_1^{\text{mod}})[o/\mathbf{this}, \bar{v}/\bar{p}]) \\
& \quad \quad \quad \vdots \\
& \quad \quad \wedge \\
& \quad \quad p \neq \text{close}(\text{ev}(e_n^{\text{mod}})[o/\mathbf{this}, \bar{v}/\bar{p}]) \\
& \quad \quad \downarrow \\
& H(p) = H^{\text{old}}(p) \wedge p \notin C \wedge (p \in P) = (p \in P^{\text{old}}))
\end{aligned}$$

Definition 16. The body of a mutator method

mutator $m(\bar{p})$ **requires** e_P ; **modifies** $e_1^{\text{mod}}, \dots, e_n^{\text{mod}}$; **ensures** e_Q ; $\{\bar{s}\}$

declared by a class c is valid if the body is a valid continuation with respect to the postcondition.

$$\begin{aligned}
& \Sigma \vdash \\
& \quad \forall H, P, o, \bar{v} \bullet \\
& \quad \text{classof}(o) = c \wedge \{o, \bar{v}\} \subseteq \text{dom}(H) \cup \mathbb{B} \cup \mathbb{Z} \cup \{\text{null}\} \\
& \quad \wedge \text{reachable}(H^{\text{old}}, P^{\text{old}}, C^{\text{old}}) \wedge \text{reachable}(H, P, C) \\
& \quad \quad \wedge \\
& \quad \quad \text{close}(\text{ev}(e_P)[o/\mathbf{this}, \bar{v}/\bar{p}]) \\
& \quad \quad \downarrow \\
& \text{close}(\text{vs}(\bar{s}, \text{post}(o.m(\bar{v}))) [o/\mathbf{this}, \bar{v}/\bar{p}, H/H^{\text{old}}, P/P^{\text{old}}, C/C^{\text{old}}])
\end{aligned}$$

Definition 17. The main routine \bar{s} is valid if it is a valid continuation.

$$\Sigma \vdash \text{close}(\text{vs}(\bar{s}, \text{true})) [\emptyset/H, \emptyset/P, \emptyset/C]$$

Definition 18. A program is valid if all invariants, method headers, method bodies, derived invariants, and dynamic invariants are valid, and all classes correctly implement the interfaces mentioned in their **implements** clause, and the main routine is valid.

5 Soundness

Lemma 1. *If $R \vdash e : \text{confined}$, then the value of e depends only on the states carried by the variables in R .*

Lemma 2. *If e is a rep expression with respect to R , then the state carried by e depends only on the states carried by R .*

Lemma 3. *The theory Σ is true.*

$$\mathcal{J} \models \Sigma$$

Lemma 4. *If an expression is valid in a given context, its evaluation does not get stuck, and the value yielded equals the interpretation of the expression's value term, and the state yielded, if any, equals the interpretation of the expression's state term.*

For a stateful expression e :

$$\begin{array}{c} H, P, H', P', V, V_S \models \text{ve}(e) \\ \Downarrow \\ H, P, H', P', V, V_S \vdash \\ e \rightsquigarrow \mathcal{J}(\text{ev}(e), H, P, H', P', V, V_S), \mathcal{J}(\text{es}(e), H, P, H', P', V, V_S) \end{array}$$

For a stateless expression e :

$$\begin{array}{c} H, P, H', P', V, V_S \models \text{ve}(e) \\ \Downarrow \\ H, P, H', P', V, V_S \vdash e \rightsquigarrow \mathcal{J}(\text{ev}(e), H, P, H', P', V, V_S) \end{array}$$

Proof. By induction on the structure of e , and case analysis on the form of e . \square

Our dynamic semantics is behaviorally equivalent to that of Java, except that programs might get stuck more often. The most important peculiarity about our dynamic semantics is that the object states of the rep objects of a packed object o are cached in the state of o . However, this does not impact behavior.

Definition 19. *A program state*

$$\sigma = (H, P, V, \bar{s}, (H_1, P_1, o_1, m_1, \bar{v}_1, x_1, V_1, \bar{s}_1) \cdot \dots \cdot (H_n, P_n, o_n, m_n, \bar{v}_n, x_n, V_n, \bar{s}_n) \cdot \langle \rangle)$$

is valid if all of the following hold:

- PSV1: *each activation record's continuation's verification condition holds*

with respect to the activation record's postcondition.

$$\begin{aligned}
& \mathcal{J}, H, P, V \models \text{vc}(\bar{s}, \text{post}(\sigma)) \\
& \quad \wedge \\
& \mathcal{J}, H_1, P_1, V_1 \models \text{vcc}(\text{call } x_1 := o_1.m_1(\bar{v}_1); \bar{s}_1, \text{post}_1(\sigma)) \\
& \quad \wedge \\
& \quad \vdots \\
& \quad \wedge \\
& \mathcal{J}, H_n, P_n, V_n \models \text{vcc}(\text{call } x_n := o_n.m_n(\bar{v}_n); \bar{s}_n, \text{post}_n(\sigma))
\end{aligned}$$

- PSV2: each packed object is valid

$$\forall o \in P \bullet \text{valid}(o, H(o))$$

- PSV3: the object states cached in a packed object are up-to-date

$$\forall o \in P, f \in \text{reffield}(\text{classof}(o)) \bullet H(o)(f_{\text{state}}) = H(H(o)(f))$$

Lemma 5. *A valid program's initial state is a valid state.*

Proof. Follows immediately from validity of the main routine. \square

Lemma 6. *Execution steps of a valid program transform valid states to valid states.*

Proof. Prove that the execution step preserves the validity of each activation record, by case analysis on the continuation's first statement. Program state validity conditions PSV2 and PSV3 are established by **pack** operations and maintained by field write operations. \square

Lemma 7. *All states reached by a valid program are valid states.*

Proof. By induction on the length of the execution. Combines Lemma 5 and Lemma 6. \square

Lemma 8. *Valid states are not stuck.*

Proof. By case analysis on the first statement of the top activation record's continuation. Uses Lemma 4. \square

Theorem 1. *Valid programs do not get stuck.*

Proof. By combining Lemma 7 and Lemma 8. \square

6 Examples

In this section we show a number of example programs.

The operators used in the examples are as follows. Note: the interpretation of an operator for an argument list outside of its domain is `null`.

Name	Syntax	Domain	Interpretation
<code>true</code>	<code>true</code>	$\{()\}$	$() \mapsto \text{true}$
<code>eq</code>	$e_1 = e_2$	$\mathcal{V} \times \mathcal{V}$	$(v_1, v_2) \mapsto (v_1 = v_2)$
<code>not</code>	$\neg e$	\mathbb{B}	$(b) \mapsto (\neg b)$
<code>and</code>	$e_1 \wedge e_2$	$\mathbb{B} \times \mathbb{B}$	$(b_1, b_2) \mapsto (b_1 \wedge b_2)$
<code>instanceof_c</code>	$e \text{ instanceof } c$	\mathcal{V}	$(v) \mapsto (v \in \mathcal{O} \wedge \text{classof}(v) = c)$

Note: the example in Figures 13 and 14 uses a set-valued rep field *perms*; this feature is not part of the formalized system. Also, classes *Map* and *MapFactory* are omitted.

```

class Cell implements {
  x;
  rep;
  invariant true;
  inspector getX()
    reads;
    requires true;
    { return this.state.x; }
  derived_invariant true;
  mutator setX(value)
    requires ¬this.committed ∧ this.inv;
    modifies this;
    ensures ¬this.committed ∧ this.inv
      ∧ heap(this).getXCell() = value;
  {
    unpackCell this;
    this.x := value;
    packCell this;
    return null;
  }
}

class CellFactory implements {
  rep;
  invariant true;
  derived_invariant true;
  mutator createCell(value)
    requires true;
    modifies;
    ensures result instanceof Cell
      ∧ ¬result.committed ∧ result.inv
      ∧ heap(result).getXCell() = value;
  {
    c := new Cell;
    c.x := value;
    packCell c;
    return c;
  }
}

cf := new CellFactory;
y := 0;
call c1 := cf.createCell(y);
y := heap(c1).getXCell();
if (y = 0) { } else { y := null.getXCell(); }
y := 5;
call c2 := cf.createCell(y);
y := 10;
call c1.setX(y);
if (heap(c1).getXCell() = 10) { }
else { y := heap(null).getXCell(); }
if (heap(c2).getXCell() = 5) { }
else { y := heap(null).getXCell(); }
return null;

```

Figure 6: A class specified using an inspector method, and a client program

```

interface Array {
  inspector getLength() reads; requires true;
  inspector get(index)
    reads;
    requires  $\text{isint}(\textit{index}) \wedge 0 \leq \textit{index} \wedge \textit{index} < \text{this}_{\text{state}}.\textit{getLength}_{\text{Array}}()$ ;
  dynamic_invariant  $\text{isint}(\text{this}_{\text{state}}.\textit{getLength}_{\text{Array}}()) \wedge 0 \leq \text{this}_{\text{state}}.\textit{getLength}_{\text{Array}}()$ ;
  mutator set(index, value)
    requires  $\text{isint}(\textit{index}) \wedge 0 \leq \textit{index} \wedge \textit{index} < \text{heap}(\text{this}).\textit{getLength}()$ 
       $\wedge \neg \text{this}.\textit{committed} \wedge \text{this}.\textit{inv}$ ;
    modifies this;
    ensures  $\neg \text{this}.\textit{committed} \wedge \text{this}.\textit{inv}$ 
       $\wedge \text{forall}\{i \text{ in } (0 : \text{heap}(\text{this}).\textit{getLength}_{\text{Array}}())$ ;
         $\text{heap}(\text{this}).\textit{get}_{\text{Array}}(i) = (i = \textit{index} ? \textit{value} : \text{old}(\text{heap}(\text{this}).\textit{get}_{\text{Array}}(i)))\}$ ;
}

class Arrayn implements Array {
  elem0;
  .
  .
  elemn-1;
  rep;
  invariant true;
  inspector getLength() reads; requires true;
  { return n; }
  inspector get(index)
    reads;
    requires  $\text{isint}(\textit{index}) \wedge 0 \leq \textit{index} \wedge \textit{index} < \text{this}_{\text{state}}.\textit{getLength}_{\text{Array}_n}()$ ;
  { return  $\textit{index} = 0 ? \text{this}_{\text{state}}.\textit{elem}_0 : \dots : \textit{index} = n - 1 ? \text{this}_{\text{state}}.\textit{elem}_{n-1} : \text{null}$ ; }
  mutator set(index, value)
    requires  $\text{isint}(\textit{index}) \wedge 0 \leq \textit{index} \wedge \textit{index} < \text{heap}(\text{this}).\textit{getLength}_{\text{Array}_n}()$ 
       $\wedge \neg \text{this}.\textit{committed} \wedge \text{this}.\textit{inv}$ ;
    modifies this;
    ensures  $\neg \text{this}.\textit{committed} \wedge \text{this}.\textit{inv}$ 
       $\wedge \text{heap}(\text{this}).\textit{getLength}_{\text{Array}_n}() = \text{old}(\text{heap}(\text{this}).\textit{getLength}_{\text{Array}_n}())$ 
       $\wedge \text{forall}\{i \text{ in } (0 : \text{heap}(\text{this}).\textit{getLength}_{\text{Array}_n}())$ ;
         $\text{heap}(\text{this}).\textit{get}_{\text{Array}_n}(i) = (i = \textit{index} ? \textit{value} : \text{old}(\text{heap}(\text{this}).\textit{get}_{\text{Array}_n}(i)))\}$ ;
  {
    unpackArrayn this;
    if ( $\textit{index} = 0$ ) { this.elem0 := value; } else { }
    .
    .
    if ( $\textit{index} = n - 1$ ) { this.elemn-1 := value; } else { }
    packArrayn this;
    return null;
  }
}

```

Figure 7: Array example

```

class ArrayFactory implements {
  rep;
  invariant true;
  mutator createArray(length)
    requires isint(length)  $\wedge$   $0 \leq \text{length}$ ;
    modifies;
    ensures result instanceof Array  $\wedge$   $\neg$ result.committed  $\wedge$  result.inv;
       $\wedge$  heap(result).getLengthArray() = length;
  {
    if (length = 0) { r := new Array0; packArray0 r; } else { }
    :
    if (length = M) { r := new ArrayM; packArrayM r; } else { }
    if (length > M) { call d := this.createArray(length); } else { }
    return r;
  }
  mutator arraycopy(src, srcPos, dest, destPos, length)
    requires src instanceof Array  $\wedge$  isint(srcPos)  $\wedge$  dest instanceof Array  $\wedge$  isint(destPos)
       $\wedge$  isint(length)
       $\wedge$   $\neg$ src.committed  $\wedge$  src.inv
       $\wedge$   $\neg$ dest.committed  $\wedge$  dest.inv
       $\wedge$   $0 \leq \text{srcPos} \wedge 0 \leq \text{length} \wedge 0 \leq \text{destPos}$ 
       $\wedge$   $\text{srcPos} + \text{length} \leq \text{heap}(src).getLength_{Array}()$ 
       $\wedge$   $\text{destPos} + \text{length} \leq \text{heap}(dest).getLength_{Array}()$ ;
    modifies dest;
    ensures  $\neg$ dest.committed  $\wedge$  dest.inv
       $\wedge$  heap(dest).getLengthArray() = old(heap(dest).getLengthArray())
       $\wedge$  forall{i in (0 : heap(dest).getLengthArray())}
        heap(dest).getArray(i) =
          (destPos  $\leq$  i  $\wedge$  i < destPos + length ?
            old(heap(src).getArray(i - destPos + srcPos)) : old(heap(dest).getArray(i)));
  {
    if (0 < length) {
      if (srcPos < destPos) {
        i := destPos + length - 1;
        v := heap(src).get(srcPos + length - 1);
        call r := dest.set(i, v);
        length := length - 1;
        call r := this.arraycopy(src, srcPos, dest, destPos, length);
      } else {
        v := heap(src).getArray(srcPos);
        call r := dest.set(destPos, v);
        srcPos := srcPos + 1; destPos := destPos + 1; length := length - 1;
        call r := this.arraycopy(src, srcPos, dest, destPos, length);
      }
    } else { }
    return null;
  }
}

```

Figure 8: Encoding of arrays in the formal program syntax of this paper. The program contains a class Array_n for each n where $0 \leq n \leq M$. If the argument to a call of method createArray is greater than M , the call does not return.

```

class ArrayList implements {
  elems;
  count;
  rep elems;
  invariant this.state.elems instanceof Array  $\wedge$  isint(this.state.count)
     $\wedge$   $0 \leq$  this.state.count  $\wedge$  this.state.count  $\leq$  this.state.elems.state.getLength_Array();
  inspector getCount() reads; requires true;
  { return this.state.count; }
  inspector getItem(index)
    reads;
    requires isint(index)  $\wedge$   $0 \leq$  index  $\wedge$  index  $<$  this.state.getCount_ArrayList();
  { return this.state.elems.state.get_Array(index); }
  derived_invariant isint(this.state.getCount_ArrayList())  $\wedge$   $0 \leq$  this.state.getCount_ArrayList();
  mutator add(x)
    requires  $\neg$ this.committed  $\wedge$  this.inv;
    modifies this;
    ensures  $\neg$ this.committed  $\wedge$  this.inv
       $\wedge$  heap(this).getCount_ArrayList() = old(heap(this).getCount_ArrayList()) + 1
       $\wedge$  foralli in (0 : old(heap(this).getCount_ArrayList()));
        heap(this).getItem_ArrayList(i) = old(heap(this).getItem_ArrayList(i))
       $\wedge$  heap(this).getItem_ArrayList(old(heap(this).getCount_ArrayList())) = x;
  {
    unpack_ArrayList this;
    e := this.elems_heap;
    n := this.count_heap;
    this.count := n + 1;
    if (heap(e).getLength_Array()  $<$  this.count_heap) {
      f := new ArrayFactory;
      c := 2  $\times$  this.count_heap;
      call a := f.createArray(c);
      z := 0;
      call r := f.arraycopy(e, z, a, z, n);
      this.elems := a;
      e := a;
    } else { }
    call r := e.set(n, x);
    return null;
  }
}

```

Figure 9: ArrayList example

```

class ArrayListFactory implements {
  rep;
  invariant true;
  derived invariant true;
  mutator createArrayList(xs)
    requires xs instanceof Array  $\wedge$   $\neg$ xs.committed  $\wedge$  xs.inv;
    modifies;
    ensures result instanceof ArrayList  $\wedge$   $\neg$ result.committed  $\wedge$  result.inv
       $\wedge$  heap(result).getCountArrayList() = heap(xs).getLengthArray()
       $\wedge$  forall{i in (0 : heap(xs).getLengthArray())}
        heap(result).getItemArrayList(i) = heap(xs).getArray(i);
    {
      r := new ArrayList;
      n := heap(xs).getLengthArray();
      f := new ArrayFactory;
      call e := f.createArray(n);
      z := 0;
      call d := f.arraycopy(xs, z, e, z, n);
      r.elems := e;
      r.count := n;
      packArrayList r;
      return r;
    }
}

af := new ArrayFactory;
n := 3; call xs := af.createArray(n);
i := 0; v := 1; call d := xs.set(i, v);
i := 1; v := 2; call d := xs.set(i, v);
i := 2; v := 3; call d := xs.set(i, v);
alf := new ArrayListFactory;
call list := alf.createArrayList(xs);
i := 0; v := 5; call d := xs.set(i, v);
if (heap(list).getItemArrayList(0) = 1) { } else { call d := null.createArray(x); }

```

Figure 10: ArrayList example

```

class SecurityPermission implements Permission {
  flags;
  rep;
  invariant isint(this.state.flags);
  inspector getFlags() reads; requires true; { return this.state.flags; }
  inspector isSubsetOf(other) reads other;
  requires isobj(other) ∧ classof(other) = classof(this); {
    return (this.state.flags & ~other.state.flags) = 0;
  }
  derived_invariant
  isint(this.state.getFlagsSecurityPermission())
  ∧ forall{p in object.stateSecurityPermission;
    this.state.isSubsetOfSecurityPermission(Pstate) =
    ((this.state.getFlagsSecurityPermission() & ~pstate.getFlagsSecurityPermission()) = 0)};
  mutator setFlags(flags)
  requires ¬this.committed ∧ this.inv ∧ isint(flags);
  modifies this;
  ensures ¬this.committed ∧ this.inv ∧ heap(this).getFlagsSecurityPermission() = flags;
  { unpackSecurityPermission this; this.flags := flags; packSecurityPermission this; }
  mutator copy()
  requires ¬this.committed ∧ this.inv;
  modifies;
  ensures isobj(result) ∧ classof(result) = classof(this)
  ∧ forall{p in object.state∞; classof(p) = classof(this) ⇒
    heap(checkvirtual(result)).isSubsetOfSecurityPermission(p) =
    heap(this).isSubsetOfSecurityPermission(p)};
  {
    c := new SecurityPermission;
    f := this.flagsheap;
    c.flags := f;
    packSecurityPermission c;
    return c;
  }
}
class SecurityPermissionFactory implements {
  rep;
  invariant true;
  derived_invariant true;
  mutator createSecurityPermission(flags)
  requires isint(flags);
  modifies;
  ensures result instanceof SecurityPermission ∧ ¬result.committed ∧ result.inv
  ∧ heap(result).getFlagsSecurityPermission() = flags;
  {
    p := new SecurityPermission;
    p.flags := flags;
    packSecurityPermission p;
    return p;
  }
}
interface Permission {
  inspector isSubsetOf(other) reads other;
  requires isobj(other) ∧ classof(other) = classof(this);
  dynamic_invariant true;
  mutator copy()
  requires ¬this.committed ∧ this.inv;
  modifies;
  ensures isobj(result) ∧ classof(result) = classof(this)
  ∧ forall{p in object.state∞; classof(p) = classof(this) ⇒
    heap(checkvirtual(result)).isSubsetOfPermission(p) =
    heap(this).isSubsetOfPermission(p)};
}

```

Figure 11: Permission example

```

class PermissionCell implements {
  permission;
  rep permission;
  invariant this.state.permission instanceof Permission;
  inspector contains(p) reads p; requires p instanceof Permission;
  {
    return classof(this.state.permission) = classof(p)
      ? p.state.isSubsetOfPermission(this.state.permission.state) : false;
  }
}
class PermissionCellFactory implements {
  rep;
  invariant true;
  derived_invariant true;
  mutator createPermissionCell(p)
  requires p instanceof Permission ∧ ¬p.committed ∧ p.inv;
  modifies;
  ensures result instanceof PermissionCell ∧ ¬result.committed ∧ result.inv
    ∧ forall{q in object.state∞; q instanceof Permission ⇒
      this.state.containsPermissionCell(q.state) =
        (classof(q) = classof(p) ∧ q.state.isSubsetOfPermission(p.state))};
  {
    c := p.copy();
    cell := new PermissionCell;
    cell.permission := c;
    packPermissionCell cell;
    return cell;
  }
}
spf := new SecurityPermissionFactory;
f := 1;
call p := spf.createSecurityPermission(f);
cf := new PermissionCellFactory;
call c := cf.createPermissionCell(p);
if (heap(c).containsPermissionCell(heap(p))) { } else { call r := null.createPermissionCell(); }
f := 3;
call r := p.setFlagsPermission(f);
if (¬heap(c).containsPermissionCell(heap(p))) { } else { call r := null.createPermissionCell(); }

```

Figure 12: PermissionCell example

```

class PermissionSet implements {
  map, perms;
  rep map, perms;
  invariant this.state.map instanceof Map
  ^ forall{k in value;
    this.state.map.state.contains_Map(k) =>
      classof(this.state.map.state.get_Map(k)) = k
      ^ this.state.map.state.get_Map(k) ∈ this.state.perms};
  inspector contains(p) reads p; requires p instanceof Permission;
  {
    return this.state.map.state.contains_Map(classof(p))
      ? p.state.isSubsetOf_Permission(this.state.perms.state[this.state.map.state.get_Map(classof(p))])
      : false;
  }
  mutator setPermission(p)
  requires ¬this.committed ^ this.inv
  ^ p instanceof Permission ^ ¬p.committed ^ p.inv;
  modifies this;
  ensures ¬this.committed ^ this.inv
  ^ forall{q in object.state_∞; q instanceof Permission =>
    heap(this).contains_PermissionSet(q.state)
    = (classof(q) = classof(p) ? q.state.isSubsetOf_Permission(p.state)
    : old(this.state.contains_PermissionSet(q)));
  {
    unpack_PermissionSet this;
    if (heap(this.map_heap).contains_Map(classof(p))) {
      s := this.perms_heap \ {p};
      this.perms := s;
    }
    call c := p.copy();
    s := this.perms ∪ {c};
    this.perms := s;
    m := this.map;
    call r := m.set(classof(p), c);
    pack_PermissionSet this;
  }
}
class PermissionSetFactory implements {
  rep;
  invariant true;
  derived_invariant true;
  mutator createPermissionSet()
  requires true;
  modifies;
  ensures result instanceof PermissionSet ^ ¬result.committed ^ result.inv
  ^ forall{p in object.state_∞; result.state.contains_PermissionSet(p.state) = false;
  {
    s := new PermissionSet;
    mf := new MapFactory;
    m := mf.createMap();
    s.map := m;
    p := ∅;
    s.perms := p;
    pack_PermissionSet s;
    return s;
  }
}

```

Figure 13: PermissionSet example

```
sf := new PermissionSetFactory;
call s := sf.createPermissionSet();
spf := new SecurityPermissionFactory;
f := 1;
call p := spf.createSecurityPermission(f);
call r := s.setPermission(p);
if (heap(s).containsPermissionSet(heap(p))) { } else { call r := null.createPermissionSet(); }
f := 3;
call r := p.setFlags(f);
if (-heap(s).containsPermissionSet(heap(p))) { } else { call r := null.createPermissionSet(); }
```

Figure 14: PermissionSet client