

Sound reasoning about unchecked exceptions: Soundness proof (Draft)

Bart Jacobs Peter Müller Frank Piessens

April 18, 2007

Abstract

In this note we formalize a multithreaded Java-like programming language with unchecked exceptions, try-catch blocks, and synchronized blocks, as well as method contracts. We further formalize our verification condition generation-based modular static verification approach that verifies that the program complies with its method contracts even in the presence of unchecked exceptions, and prove its soundness.

1 Programming model

1.1 Programs

To formalize the rules imposed by our programming model, we first define a small language consisting of a subset of Java (minus static typing) plus two kinds of annotations (indicated by the gray background): share statements and method contracts. Its syntax is shown in Figure 1. An example program in this language is shown in Figure 2. We discuss the language and define *well-formedness* of programs.

A program π consists of a number of classes and interfaces and a main routine. A class may declare zero or more fields, and both classes and interfaces may declare zero or more methods. Interface methods consist of a header only, whereas class methods consist of a header and a body. Each method header includes a method contract, consisting of a *precondition* (**requires** clause) and a *postcondition* (**ensures** clause). The precondition and postcondition specify an assertion that must hold in the pre-state and the post-state of method calls, respectively. Since method contracts are used only for verifying modularly whether a given program complies with the programming model, and are not part of the programming model itself, it is safe to ignore them in this section.

In addition to method contracts, the syntax supports a second type of annotations, namely *share statements*. Using share statements, a programmer indicates when an object transitions from unshared to shared.

The formalization does not include **try-finally** statements as a separate construct. Instead, we consider them to be syntactic sugar:

$$\mathbf{try} \{ \bar{s}_1 \} \mathbf{finally} \{ \bar{s}_2 \} \equiv \mathbf{try} \{ \bar{s}_1 \} \mathbf{catch} \{ \bar{s}_2 \mathbf{throw}; \} \bar{s}_2$$

The language is not statically typed. Rather, to avoid formalizing a type system, type mismatches are considered run-time errors. The static verification

\mathcal{C}	class names	\mathcal{M}	method names	Φ	logical formulae
\mathcal{I}	interface names	\mathcal{F}	field names	$\underline{\mathcal{O}}$	object references
		\mathcal{X}	variable names	$\underline{\Theta}$	thread-relevant states

$$C \in \mathcal{C}, I \in \mathcal{I}, m \in \mathcal{M}, f \in \mathcal{F}, x \in \mathcal{X}, \varphi \in \Phi, o \in \underline{\mathcal{O}}, \theta \in \underline{\Theta}$$

```

 $\pi$  ::=  $\langle \text{iface} \mid \text{class} \rangle^* s^*$ 
iface ::= interface  $I \{ mh^* \}$ 
mh ::=  $m(x^*)$  requires  $\varphi$ ; ensures  $\varphi$ ;
v ::= null  $\mid$   $\underline{o}$ 
g ::= this  $\mid x \mid v$ 
class ::= class  $C$  implements  $I^* \{ field^* \text{ invariant } \varphi; meth^* \}$ 
field ::=  $f$ ;
meth ::=  $mh \{ s^* \}$ 
 $\tau$  ::=  $C \mid I$ 
s ::= if  $(g = g) \{ s^* \}$  else  $\{ s^* \}$  assert  $g$  instanceof  $\tau$ ;
 $\mid x := g.f; \mid g.f := g; \mid x := \text{new } C; \mid x := g.m(g^*); \mid \text{start } g.m();$ 
 $\mid \text{share } g; \mid \text{synchronized } (g) \{ s^* \} \mid \text{par } g.m() \parallel g.m();$ 
 $\mid \text{try binding } g^*; \text{invariant } \phi; \{ s^* \} \text{catch } \{ s^* \} \mid \text{throw};$ 
 $\mid \text{return } g; \mid x := \text{receive } [\theta] \underline{o.m(v^*);} \mid \text{unlock } o;$ 
 $\mid \text{receive\_par } [\theta, \text{tid}] \underline{o.m()} \parallel \underline{o.m()};$ 
 $\mid \text{pack}_C g; \mid \text{unpack}_C g;$ 

```

Figure 1: Syntax of a small Java-like language without static typing, but with two kinds of annotations (indicated by the gray background): method contracts and share statements. The underlined elements appear only as part of continuations during program execution (see Section 1.2) and are not allowed to appear in well-formed programs. The syntax of the logical formulae used in method contracts is given in Section 2.

approach detailed in Section 2 guarantees the absence of programming model violations as well as run-time errors (i.e., null dereferences and type mismatches).

Our static verification approach performs modular verification. This means that for each method, all executions of that method are considered, not just those that occur in executions of the program. For example, for method *run* of class *Session* in Figure 2, all executions that satisfy *run*'s precondition are considered, including those where the *count* field contains a null value, even though there is no execution of the program of Figure 2 where the method is called in a state where *count* is null. It follows that in the absence of the **assert** statement, the method would be considered invalid since in an execution where *count* is null, the **synchronized** statement would cause a null dereference. This limits the usefulness of the approach, since users are not interested in errors that do not occur in program executions. To alleviate this limitation, the programmer may restrict the set of executions considered by the static verification approach, by inserting **assert** statements into the program. If an **assert** statement's condition evaluates to false, the statement is said to *fail*. If in a given execution an **assert** statement fails, the execution is *stuck* (in Java, an exception is thrown), but for purposes of static verification, the execution is considered to be valid (or in other words, the execution is not considered further), since the failure is assumed to mean that the method execution never

```

class Counter { count; }

class Session {
  counter;
  run()
  requires  $L_t = \emptyset \wedge \text{this} \in A$ ;
  {
    /* wait for event */;
    c := this.counter;
    assert c instanceof Counter;
    synchronized (c) { n := c.count; c.count := n + 1; }
    this.run();
  }
}

c := new Counter;
share c;
s := new Session; s.counter := c; start s.run();
s := new Session; s.counter := c; start s.run();

```

Figure 2: An example program in the formal syntax of Figure 1. (Note: the example also uses integer values and integer operations; these are omitted from the formal development for simplicity.)

appears in an execution of the whole program. Other verification approaches outside the scope of this thesis, such as code review, model checking, or testing, may be used to verify such assumptions. (Note that the **assert** statements of this chapter correspond to the **assume** statements of Spec# [?]. In this chapter we use **assert** syntax since this is the existing syntax of the Java language.)

We only consider *well-formed* programs. Well-formed programs have no name clashes. Also, local variables need not be declared before they are assigned (in fact, our language does not have local variable declarations) but they must be assigned before they are used. By requiring that both branches of a conditional statement assign to the same variables, we can define a notion of *free variables* at each program point by considering an assignment to *bind* the variable occurrences that occur after it in the control flow and that are not hidden by a later assignment. Further, a well-formed program must not contain any of the syntactic forms that are intended to appear only during program execution (i.e., the constructs underlined in Figure 1). Lastly, classes must implement all methods of their declared interfaces, and if a method is used to start a new thread, its precondition must be as prescribed.

Definition 1. A program π is well-formed if all of the following hold:

- No two interfaces have the same name. No two classes have the same name. No interface has the same name as a class. No two parameters of a given method have the same name. No two fields have the same name (even if they appear in different classes). No two interface methods have the same name (even if they appear in different interfaces). If two class methods have the same name m , then the methods are in different classes and the program declares an interface I that declares a method with name m and both classes implement interface I .

- If one branch of an **if** statement contains an assignment to a variable x , then so does the other branch.
- If a statement uses a variable x , then an assignment to x appears before the statement in the method body (ignoring the other branch of an enclosing **if** statement), or x is a parameter of the enclosing method or **this**.
- The last statement of a method body and of the main routine is a **return** statement and a **return** statement does not appear anywhere else.
- The program does not contain any **receive** or **unlock** statements or object references.
- If a class implements an interface I and this interface declares a method with name m then the class declares a method with name m and its header is identical to the header of the method named m declared by interface I .
- Each class name, interface name, field name, and method name that appears in the program is declared by the program.
- The number of arguments specified in a call equals the number of parameters declared by the corresponding method.
- The only free variables in an object invariant are **this** and H . H occurs only in terms of the form $H[\mathbf{this}, f]$.
- If a method is mentioned in a **start** statement, then its requires clause is exactly $L_t = \emptyset \wedge \mathbf{this} \in A$. (Note: the semantics of method contracts is defined in Section 2.)

Notice that the example program of Figure 2 is well-formed.

All concepts in the remainder of this paper are implicitly parameterized by a program π .

1.2 Program executions

We now formalize the semantics of our annotated Java subset. While remaining faithful to Java semantics, our semantics additionally tracks the extra state variables (specifically, access sets and shared sets) required by the programming model. We also define a set of *legal program states*. A program state is legal if all thread states are legal. A thread is in a legal state if it is not about to violate the programming model or cause a run-time error (in particular, a null dereference or a type mismatch). In Section 1.3, we show that programs that reach only legal states are data-race-free. Note: we define legality as a separate judgment rather than encoding it as absence of progress (i.e., thread execution getting stuck) since in concurrent executions, on the one hand stuckness of a thread might not be due to an error in that thread (for example, if the thread is waiting for a lock that is never released), and on the other hand a thread that is stuck due to an error might become un-stuck again as a result of actions of other threads (for example, if the thread is attempting to lock an unshared object and the object is subsequently shared by another thread).

We use the following notation. \mathcal{T} denotes the set of thread identifiers. Furthermore, v represents a value (i.e. $v \in \mathcal{O} \cup \{\text{null}\}$) and o represents an

object reference (i.e., a non-null value). $\text{fields}(C)$ represents the set of fields declared by class C . We use $C \prec I$ to denote that class C implements interface I . $\text{mbody}(C, m)$ denotes the body of method m declared in class C , and program_main denotes the program's main routine. $\text{declaringClass}(f)$ denotes the name of the class that declares field f , and $\text{declaringType}(m)$ denotes the name of the interface that declares method m , or the name of the class that declares m if no interface declares a method m . Also, we assume the existence of a function classof that maps object references to class names. We assume that for each class name $C \in \mathcal{C}$, there are infinitely many object references $o \in \mathcal{O}$ such that $\text{classof}(o) = C$. $\text{objectRefs}(\phi)$ and $\text{free}(\phi)$ denote the free object references and free variables, respectively, in syntactic entity ϕ . We use $f[x \mapsto y]$ to denote the update of the function f at argument x with value y . Specifically, $f[x \mapsto y](z)$ equals y if $z = x$ and $f(z)$ otherwise. Similarly, $f \setminus \{(x, y)\}$ removes the mapping of x to y from f , and thereby removes x from the domain of f . We use the notation $\bar{s}[v/x]$ to denote substitution of a value v for a program variable x in a thread continuation (i.e., list of statements) \bar{s} . This substitution replaces only the free occurrences of x , i.e., the ones that are not bound by an assignment inside \bar{s} . We denote the empty list as ϵ and a list with head h and tail t as $h \cdot t$.

The dynamic semantics is defined as a small step relation on *program states*.

Definition 2. A program state

$$\sigma = (\text{H}, \text{L}, \text{S}, \text{T})$$

consists of:

- the heap H , a partial function that maps object references to object states. An object state is a partial function that maps field names to values.

$$\text{H} : \mathcal{O} \hookrightarrow (\mathcal{F} \hookrightarrow \mathcal{O} \cup \{\text{null}\})$$

The domain of H consists of all allocated objects. The domain of an object state $\text{H}(o)$ consists of the declared fields of the class $\text{classof}(o)$ of o .

- the lock map L , a partial function that maps a locked object to the identifier of the thread that holds the lock

$$\text{L} : \mathcal{O} \hookrightarrow \mathcal{T}$$

- the shared set S , the set of shared objects
- the thread set T . Each thread state $(\text{tid}, \text{A}, \text{F}) \in \text{T}$ consists of a unique thread identifier $\text{tid} \in \mathcal{T}$, an access set $\text{A} \subseteq \mathcal{O}$ and a list of activation records $\text{F} = \bar{R}$. An activation record $R = (\bar{s}, \bar{b}) \in s^* \times (\mathcal{P}(\mathcal{O}) \times c \times \Phi)^*$ consists of an active continuation and a list of enclosing blocks $\bar{b} = (B, c)$, consisting of a set B of block-bound objects, and a block continuation c whose syntax is given by

$$c ::= \mathbf{unlock} \ o; \ s^* \mid \mathbf{catch} \ [\theta, \{x^*\}, \phi] \ \{s^*\} \ s^*$$

We shall sometimes use uncurried syntax for the heap: $\text{H}(o, f)$ is shorthand for $\text{H}(o)(f)$, and $\text{H}[(o, f) \mapsto v]$ is shorthand for $\text{H}[o \mapsto \text{H}(o)[f \mapsto v]]$.

$$\begin{array}{c}
\text{[LEGAL-IF]} \quad \mathbf{H, L, S} \vdash (\text{tid}, \mathbf{A}, (V, \mathbf{if} (g_1 = g_2) \{ \bar{s}_1 \} \mathbf{else} \{ \bar{s}_2 \} \bar{s}, \bar{b}) \cdot \mathbf{F}) : \text{legal} \\
\text{[IF]} \quad \frac{V(g_1) = V(g_2) \Rightarrow \bar{s}' = \bar{s}_1 \quad V(g_1) \neq V(g_2) \Rightarrow \bar{s}' = \bar{s}_2}{(\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, \mathbf{if} (g_1 = g_2) \{ \bar{s}_1 \} \mathbf{else} \{ \bar{s}_2 \} \bar{s}, \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, \bar{s}', \bar{s}, \bar{b}) \cdot \mathbf{F}))} \\
\text{[LEGAL-ASSERT]} \quad \mathbf{H, L, S} \vdash (\text{tid}, \mathbf{A}, (V, \mathbf{assert} \ g \ \mathbf{instanceof} \ \tau; \bar{s}, \bar{b}) \cdot \mathbf{F}) : \text{legal} \\
\text{[ASSERT]} \quad \frac{V(g) \neq \text{null} \quad \text{classof}(V(g)) \preceq \tau}{(\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, \mathbf{assert} \ g \ \mathbf{instanceof} \ \tau; \bar{s}, \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, \bar{s}, \bar{b}) \cdot \mathbf{F}))} \\
\text{[LEGAL-READ]} \quad \frac{V(g) \neq \text{null} \quad V(g) \in \mathbf{A} \quad \text{classof}(V(g)) = \text{declaringClass}(f)}{\mathbf{H, L, S} \vdash (\text{tid}, \mathbf{A}, (V, x := g.f; \bar{s}, \bar{b}) \cdot \mathbf{F}) : \text{legal}} \\
\text{[READ]} \quad (\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, x := g.f; \bar{s}, \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V[x \mapsto \mathbf{H}(V(g), f)], \bar{s}, \bar{b}) \cdot \mathbf{F})) \\
\text{[LEGAL-WRITE]} \quad \frac{V(g_1) \neq \text{null} \quad V(g_1) \in \mathbf{A} \quad c(V(g_1)) = \text{declaringClass}(f)}{\mathbf{H, L, S} \vdash (\text{tid}, \mathbf{A}, (V, g_1.f := g_2; \bar{s}, \bar{b}) \cdot \mathbf{F}) : \text{legal}} \\
\text{[WRITE]} \quad (\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, g_1.f := g_2; \bar{s}, \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H}[(V(g_1), f) \mapsto V(g_2)], \mathbf{L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, \bar{s}, \bar{b}) \cdot \mathbf{F})) \\
\text{[LEGAL-NEW]} \quad \mathbf{H, L, S} \vdash (\text{tid}, \mathbf{A}, (V, x := \mathbf{new} \ C; \bar{s}, \bar{b}) \cdot \mathbf{F}) : \text{legal} \\
\text{[NEW]} \quad \frac{o \notin \text{dom}(\mathbf{H}) \quad \text{classof}(o) = C \quad \text{fields}(C) = \{f_1, \dots, f_n\}}{(\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, x := \mathbf{new} \ C; \bar{s}, \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H}[o \mapsto \{(f_1, \text{null}), \dots, (f_n, \text{null})\}], \mathbf{L, S, T} \triangleleft (\text{tid}, \mathbf{A} \cup \{o\}, (V[x \mapsto o], \bar{s}, \bar{b}) \cdot \mathbf{F}))} \\
\text{[LEGAL-SHARE]} \quad \frac{V(g) \neq \text{null} \quad V(g) \in \mathbf{A} \quad V(g) \notin \mathbf{S}}{\mathbf{H, L, S} \vdash (\text{tid}, \mathbf{A}, (V, \mathbf{share} \ g; \bar{s}, \bar{b}) \cdot \mathbf{F}) : \text{legal}} \\
\text{[SHARE]} \quad (\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, \mathbf{share} \ g; \bar{s}, \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H, L, S} \cup \{v\}, \mathbf{T} \triangleleft (\text{tid}, \mathbf{A} \setminus \{V(g)\}, (V, \bar{s}, \bar{b}) \cdot \mathbf{F})) \\
\text{[LEGAL-SYNCHRONIZED]} \quad \frac{V(g) \neq \text{null} \quad V(g) \notin \mathbf{L}^{-1}(\text{tid}) \quad V(g) \in \mathbf{S}}{\mathbf{H, L, S} \vdash (\text{tid}, \mathbf{A}, (V, \mathbf{synchronized} \ (g) \ \{ \bar{s}' \} \bar{s}, \bar{b}) \cdot \mathbf{F}) : \text{legal}} \\
\text{[SYNCHRONIZED]} \quad \frac{V(g) \notin \text{dom}(\mathbf{L})}{(\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, \mathbf{synchronized} \ (g) \ \{ \bar{s}' \} \bar{s}, \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H, L}[V(g) \mapsto \text{tid}], \mathbf{S, T} \triangleleft (\text{tid}, \mathbf{A} \cup \{V(g)\}, (V, \bar{s}', \{V(g)\}, \mathbf{unlock} \ V(g); \bar{s}) \cdot \bar{b}) \cdot \mathbf{F}))} \\
\text{[LEGAL-UNLOCK]} \quad \frac{o \in \mathbf{A} \quad (o, \text{tid}) \in \mathbf{L}}{\mathbf{H, L, S} \vdash (\text{tid}, \mathbf{A}, (V, \epsilon, (B, \mathbf{unlock} \ o; \bar{s}) \cdot \bar{b}) \cdot \mathbf{F}) : \text{legal}} \\
\text{[UNLOCK]} \quad (\mathbf{H, L, S, T} \triangleleft (\text{tid}, \mathbf{A}, (V, \epsilon, (B, \mathbf{unlock} \ o; \bar{s}) \cdot \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H, L} \setminus \{(o, \text{tid})\}, \mathbf{S, T} \triangleleft (\text{tid}, \mathbf{A} \setminus \{o\}, (V, \bar{s}, \bar{b}) \cdot \mathbf{F}))}
\end{array}$$

Figure 3: Legal thread states and execution steps. ($\mathbf{T} \triangleleft t = \mathbf{T} \cup \{t\}$) (Continued in Figure 4.)

$$\begin{array}{c}
\text{[LEGAL-CALL]} \frac{V(g) \neq \text{null} \quad \text{classof}(V(g)) \preceq \text{declaringType}(m)}{\text{H, L, S} \vdash (\text{tid}, \text{A}, (V, x := g.m(\bar{g}); \bar{s}, \bar{b}) \cdot \text{F}) : \text{legal}} \\
\text{[CALL]} \frac{\bar{s}' = \text{mbody}(\text{classof}(V(g)), m) \quad t = (\text{tid}, \text{A}, (V, x := g.m(\bar{g}); \bar{s}, \bar{b}) \cdot \text{F})}{\begin{array}{l} (\text{H, L, S, T} \triangleleft t) \\ \rightarrow (\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (\emptyset[\text{this} \mapsto V(g), \bar{x} \mapsto V(\bar{g})], \bar{s}', \epsilon) \\ \cdot (V, x := \text{receive} [(\text{H, L}^{-1}(\text{tid}), \text{S, A, B}(t))] V(g).m(V(\bar{g})); \bar{s}, \bar{b}) \cdot \text{F})) \end{array}} \\
\text{[LEGAL-RETURN]} \text{H, L, S} \vdash (\text{tid}, \text{A}, (V, \text{return } g; , \epsilon) \cdot \text{F}) : \text{legal} \\
\text{[RETURN]} \frac{(\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \text{return } g; , \epsilon) \cdot (V', x := \text{receive} \dots; \bar{s}, \bar{b}) \cdot \text{F}))}{\rightarrow (\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V'[x \mapsto V(g)], \bar{s}, \bar{b}) \cdot \text{F}))} \\
\text{[LEGAL-NEWTREAD]} \frac{V(g) \neq \text{null} \quad V(g) \in \text{A} \quad \text{classof}(V(g)) \preceq \text{declaringType}(m)}{\text{H, L, S} \vdash (\text{tid}, \text{A}, (V, \text{start } g.m(); \bar{s}, \bar{b}) \cdot \text{F}) : \text{legal}} \\
\text{[NEWTREAD]} \frac{\forall (\text{tid}'', \rightarrow, \rightarrow) \in \text{T} \bullet \text{tid}' \neq \text{tid}'' \quad \text{tid}' \neq \text{tid} \quad = \text{mbody}(\text{classof}(V(g)), m)}{\begin{array}{l} (\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \text{start } g.m(); \bar{s}, \bar{b}) \cdot \text{F})) \\ \rightarrow (\text{H, L, S, T} \triangleleft (\text{tid}', \{V(g)\}, (\emptyset[\text{this} \mapsto V(g)], \bar{s}', \epsilon) \cdot \epsilon) \triangleleft (\text{tid}, \text{A} \setminus \{V(g)\}, (V, \bar{s}, \bar{b}) \cdot \text{F})) \end{array}} \\
\text{[LEGAL-TRY-ENTER]} \text{H, L, S} \vdash \\
(\text{tid}, \text{A}, (V, \text{try binding } \bar{g}; \text{invariant } \phi; \{ \bar{s}_1 \} \text{ catch } \{ \bar{s}_2 \} \bar{s}, \bar{b}) \cdot \text{F}) : \text{legal} \\
\text{[TRY-ENTER]} \frac{\begin{array}{l} t = (\text{tid}, \text{A}, (V, \text{try binding } \bar{g}; \text{invariant } \phi; \{ \bar{s}_1 \} \text{ catch } \{ \bar{s}_2 \} \bar{s}, \bar{b}) \cdot \text{F}) \\ \bar{x} \text{ are the target variables of the assignments in } \bar{s}_1 \quad \theta = (\text{H, L}^{-1}(\text{tid}), \text{S, A, B}(t)) \end{array}}{(\text{H, L, S, T} \triangleleft t) \rightarrow (\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \bar{s}_1, (\{V(\bar{g})\}, \text{catch } [\theta, \{\bar{x}\}, \phi] (\{ \bar{s}_2 \} \bar{s}) \cdot \bar{b})) \cdot \text{F}))} \\
\text{[LEGAL-TRY-EXIT]} \text{H, L, S} \vdash (\text{tid}, \text{A}, (V, \epsilon, (B, \text{catch } [\theta, \{\bar{x}\}, \phi] \{ \bar{s}_c \} \bar{s}_n) \cdot \bar{b}) \cdot \text{F}) : \text{legal} \\
\text{[TRY-EXIT]} \frac{(\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \epsilon, (B, \text{catch } [\theta, \{\bar{x}\}, \phi] \{ \bar{s}_c \} \bar{s}_n) \cdot \bar{b}) \cdot \text{F}))}{\rightarrow (\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \bar{s}_n, \bar{b}) \cdot \text{F}))} \\
\text{[LEGAL-THROW]} \\
\frac{\neg(\exists \bar{s}', \bar{b}', B \bullet (V, \text{throw}; \bar{s}, \bar{b}) = (\text{throw}; , (B, \text{unlock } o; \bar{s}') \cdot \bar{b}') \wedge \neg(o \in \text{A} \wedge (o, \text{tid}) \in \text{L}))}{\text{H, L, S} \vdash (\text{tid}, \text{A}, (V, \text{throw}; \bar{s}, \bar{b}) \cdot \text{F}) : \text{legal}} \\
\text{[FAIL]} \frac{\bar{s} \neq (\text{throw};)}{(\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \bar{s}, \bar{b}) \cdot \text{F})) \rightarrow (\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \text{throw}; , \bar{b}) \cdot \text{F}))} \\
\text{[FAIL-CATCH]} \frac{(\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \text{throw}; , (B, \text{catch } [\theta, \{\bar{x}\}, \phi] \{ \bar{s}_c \} \bar{s}_n) \cdot \bar{b}) \cdot \text{F}))}{\rightarrow (\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \bar{s}_c, \bar{s}_n, \bar{b}) \cdot \text{F}))} \\
\text{[FAIL-UNLOCK]} (\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \text{throw}; , (B, \text{unlock } o; \bar{s}') \cdot \bar{b}) \cdot \text{F})) \rightarrow \\
(\text{H, L} \setminus \{(o, \text{tid})\}, \text{S, T} \triangleleft (\text{tid}, \text{A} \setminus \{o\}, (V, \text{throw}; , \bar{b}) \cdot \text{F})) \\
\text{[FAIL-CALL]} \frac{(\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V, \text{throw}; , \epsilon) \cdot (V', \bar{s}', \bar{b}) \cdot \text{F}))}{\rightarrow (\text{H, L, S, T} \triangleleft (\text{tid}, \text{A}, (V', \text{throw}; , \bar{b}) \cdot \text{F}))}
\end{array}$$

Figure 4: Legal thread states and execution steps. ($\text{T} \triangleleft t = \text{T} \cup \{t\}$) (Continued from Figure 3.)

$$\begin{array}{c}
\text{[LEGAL-PAR]} \\
\frac{
\begin{array}{l}
t = (\text{tid}, \mathbf{A}, (V, \mathbf{par} \ g_1.m_1() \parallel g_2.m_2()); \bar{s}, \bar{b}) \cdot \mathbf{F} \\
V(g_1) \neq \text{null} \quad \text{classof}(V(g_1)) \preceq \text{declaringType}(m_1) \\
\mathfrak{J}, H, L^{-1}(\text{tid}), S, A, B(t) \models \text{pre}(m_1)[V(g_1)/\mathbf{this}] \quad V(g_2) \neq \text{null} \\
\text{classof}(V(g_2)) \preceq \text{declaringType}(m_2) \quad \mathfrak{J}, H, L^{-1}(\text{tid}), S, A, B(t) \models \text{pre}(m_2)[V(g_2)/\mathbf{this}] \\
A_1 = \text{interp}_{\mathfrak{J}, H, L^{-1}(\text{tid}), S, B(t)}(\mathcal{A}(\text{pre}(m_1)[V(g_1)/\mathbf{this}])) \\
A_2 = \text{interp}_{\mathfrak{J}, H, L^{-1}(\text{tid}), S, B(t)}(\mathcal{A}(\text{pre}(m_2)[V(g_2)/\mathbf{this}])) \quad A_1 \cap A_2 = \emptyset
\end{array}
}{
\mathbf{H}, \mathbf{L}, \mathbf{S} \vdash t : \text{legal}
} \\
\\
\text{[PAR]} \frac{
\begin{array}{l}
t = (\text{tid}, \mathbf{A}, (V, \mathbf{par} \ g_1.m_1() \parallel g_2.m_2()); \bar{s}, \bar{b}) \cdot \mathbf{F} \quad \forall (\text{tid}'', -, -) \in \mathbf{T} \bullet \text{tid}' \neq \text{tid}'' \\
\text{tid}' \neq \text{tid} \quad A_1 = \text{interp}_{\mathfrak{J}, H, L^{-1}(\text{tid}), S, B(t)}(\mathcal{A}(\text{pre}(m_1)[V(g_1)/\mathbf{this}])) \\
A_2 = \text{interp}_{\mathfrak{J}, H, L^{-1}(\text{tid}), S, B(t)}(\mathcal{A}(\text{pre}(m_2)[V(g_2)/\mathbf{this}])) \\
\bar{s}_1 = \text{mbody}(\text{classof}(V(g_1)), m_1) \\
\bar{s}_2 = \text{mbody}(\text{classof}(V(g_2)), m_2) \quad \theta = (\mathbf{H}, \mathbf{L}^{-1}(\text{tid}), \mathbf{S}, \mathbf{A}, \mathbf{B}(t))
\end{array}
}{
\begin{array}{l}
(\mathbf{H}, \mathbf{L}, \mathbf{S}, \mathbf{T} \triangleleft t) \rightarrow (\mathbf{H}, \mathbf{L}, \mathbf{S}, \mathbf{T} \\
\triangleleft (\text{tid}, \mathbf{A} \setminus A_2, (\emptyset[\mathbf{this} \mapsto V(g_1)], \bar{s}_1, \epsilon) \cdot (V, \mathbf{receive_par} \ [\theta, \text{tid}'] \ v_1.m_1() \parallel v_2.m_2()); \bar{s}, \bar{b}) \cdot \mathbf{F}) \\
\triangleleft (\text{tid}', A_2, (\emptyset[\mathbf{this} \mapsto V(g_2)], \bar{s}_2, \epsilon) \cdot \epsilon)
\end{array}
} \\
\\
\text{[PAR-COMPLETE]} \\
\frac{
s_1 = \mathbf{throw}; \vee s_2 = \mathbf{throw}; \Rightarrow \bar{s}' = \mathbf{throw}; \quad s_1 \neq \mathbf{throw}; \wedge s_2 \neq \mathbf{throw}; \Rightarrow \bar{s}' = \bar{s}
}{
(\mathbf{H}, \mathbf{L}, \mathbf{S}, \mathbf{T} \triangleleft (\text{tid}, \mathbf{A}_1, (V_1, s_1, \epsilon) \cdot (V, \mathbf{receive_par} \ [\theta, \text{tid}'] \ o_1.m_1() \parallel o_2.m_2()); \bar{s}, \bar{b}) \cdot \mathbf{F}) \\
\triangleleft (\text{tid}', A_2, (V_2, s_2, \epsilon) \cdot \epsilon) \rightarrow (\mathbf{H}, \mathbf{L}, \mathbf{S}, \mathbf{T} \triangleleft (\text{tid}, \mathbf{A}_1 \cup A_2, (V, \bar{s}', \bar{b}) \cdot \mathbf{F})
} \\
\\
\text{[LEGAL-PACK]} \quad \mathbf{H}, \mathbf{L}, \mathbf{S} \vdash (\text{tid}, \mathbf{A}, (V, \mathbf{pack}_C \ g; \bar{s}, \bar{b}) \cdot \mathbf{F}) : \text{legal} \\
\\
\text{[PACK]} \quad (\mathbf{H}, \mathbf{L}, \mathbf{S}, \mathbf{T} \triangleleft (\text{tid}, \mathbf{A}, (V, \mathbf{pack}_C \ g; \bar{s}, \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H}, \mathbf{L}, \mathbf{S}, \mathbf{T} \triangleleft (\text{tid}, \mathbf{A}, (V, \bar{s}, \bar{b}) \cdot \mathbf{F})) \\
\\
\text{[LEGAL-UNPACK]} \quad \mathbf{H}, \mathbf{L}, \mathbf{S} \vdash (\text{tid}, \mathbf{A}, (V, \mathbf{unpack}_C \ g; \bar{s}, \bar{b}) \cdot \mathbf{F}) : \text{legal} \\
\\
\text{[UNPACK]} \quad (\mathbf{H}, \mathbf{L}, \mathbf{S}, \mathbf{T} \triangleleft (\text{tid}, \mathbf{A}, (V, \mathbf{unpack}_C \ g; \bar{s}, \bar{b}) \cdot \mathbf{F})) \rightarrow (\mathbf{H}, \mathbf{L}, \mathbf{S}, \mathbf{T} \triangleleft (\text{tid}, \mathbf{A}, (V, \bar{s}, \bar{b}) \cdot \mathbf{F}))
\end{array}$$

Figure 5: Legal thread states and execution steps. ($\mathbf{T} \triangleleft t = \mathbf{T} \cup \{t\}$) (*Continued from Figure 4.*)

Figures 3 and 4 show the definition of the small step relation \rightarrow on program states, as well as the definition of *legality* $H, L, S \vdash t : \text{legal}$ of a thread state t with respect to heap H , lock map L , and shared set S . Legality captures the rules of the programming model, as well as absence of run-time errors (i.e., null dereferences and type mismatches).

The rule IF is standard. An **assert** statement that fails (either because the operand is null or because it is not of the specified type) causes the thread to block forever (ASSERT). For reading (READ) or writing (WRITE) a field f , the target object must be non-null, part of the current thread’s access set, and of the class that declares the field. Note that field updates change the heap: the old value of the field is replaced with the new value. When creating a new object (NEW), an unused object reference is chosen from \mathcal{O} , is inserted into the heap and all its fields are initialized to the default value null. New objects are initially only accessible to their creator and therefore the reference is added to the creating thread’s access set. A thread may share (SHARE) an unshared object in its access set. By doing so, it removes the object from its access set and adds it to the global shared set S . Shared objects may be locked (SYNCHRONIZED) provided they are not locked yet. As noted in Section ??, we consider lock re-entry to be illegal. Our method effect framing approach, described in Section 2, relies on this. For the duration of the synchronized block, the lock map L marks the current thread as holder of the lock. The object is also added to the access set. When the end of the synchronized block is reached (UNLOCK), the object must still be in the thread’s access set. At this time, the object is removed from the access set and its corresponding lock is released. Invoking a method m (CALL) within a thread t results in the addition of a new activation record to t ’s call stack. The activation record contains the body of m where all free variables (**this** and parameters) are replaced with the corresponding argument values. In the caller’s activation record, the method invocation is replaced with a **receive** statement, which keeps a record of the call’s pre-state and arguments. The operands of the **receive** statement are not used by the dynamic semantics; they are used only in the soundness proof in Section 2.5. When a method call returns (RETURN), the top activation record is popped and the return value is substituted into the caller’s activation record. A new thread (NEWTREAD) is started by performing a **start** $o.m()$; operation. Accessibility of object o is transferred from the original thread to the new one. The new thread consists of a single activation record containing the body of method m where **this** is replaced by o .

Program execution starts in an *initial state*.

Definition 3. *In a program’s initial state, there is only a single thread. It has an empty access set and it executes the main routine. Moreover, the heap, the lock map, and the shared set are all empty.*

$$\text{initial}((\emptyset, \emptyset, \emptyset, \{(\text{tid}, \emptyset, \text{program_main})\}))$$

A thread’s execution is complete when the thread’s call stack consists of a single activation record whose continuation is a **return** statement. Threads whose execution is complete are not garbage collected in our semantics, since this is irrelevant for the results of this paper.

The programs allowed by the programming model are the *legal programs*, which reach only *legal program states*.

Definition 4. A program state (H, L, S, T) is legal if all thread states are legal as per Figures 3 and 4:

$$\text{legal}(H, L, S, T) \Leftrightarrow (\forall t \in T \bullet H, L, S \vdash t : \text{legal})$$

Definition 5. A program is legal if it is well-formed and all program states reached by execution of the program are legal:

$$\text{program_legal} \Leftrightarrow \text{program_wf} \wedge (\forall \sigma_0, \sigma \bullet \text{initial}(\sigma_0) \wedge \sigma_0 \rightarrow^* \sigma \Rightarrow \text{legal}(\sigma))$$

1.3 Data-race-freedom

The dynamic semantics defined above is an *interleaving* semantics, which implies a *sequentially consistent* memory model. This means that there is a total order on all field accesses such that each read of a field $o.f$ yields the value most recently written to $o.f$ in this total order. However, the Java Language Specification, Third Edition (JLS3) [?] does not guarantee sequential consistency. In general, it allows threads to see writes performed by other threads *out of order*, which is necessary to allow efficient implementations involving optimizations at the level of the compiler, the memory hierarchy, and the processor. Still, JLS3 does guarantee sequential consistency for *correctly synchronized* programs. These are programs where all sequentially consistent executions are *data-race-free*.

In conclusion, in order for the dynamic semantics in this chapter to be sound with respect to Java and for the programs we care about, we need to prove that all executions of these programs under this semantics are data-race-free as per JLS3.

In this subsection, we show that legal programs are data-race-free, by proving that \rightarrow maintains *well-formedness* of the program state. Specifically, each execution step maintains the property that the threads' access sets and the *free set* partition the heap.

Definition 6. The free set of a program state σ consists of all shared objects that are not locked.

$$\text{Free}(\sigma) = S \setminus \text{dom}(L)$$

Definition 7. A multiset of sets S partitions a set T ($S \ll T$) if all sets in S are disjoint and their union equals T .

$$S \ll T \Leftrightarrow (\forall s_1 \in S, s_2 \in S - \{s_1\} \bullet s_1 \cap s_2 = \emptyset) \wedge \bigcup S = T$$

Execution steps maintain well-formedness of the heap, the shared set, and the lock map.

Definition 8. A heap is well-formed ($\vdash H : \text{ok}$) if objects referenced from fields are allocated.

$$\forall o \in \text{dom}(H), f \in \text{dom}(H(o)) \bullet H(o)(f) \in \text{dom}(H) \cup \{\text{null}\}$$

Definition 9. A shared set is well-formed with respect to a heap ($H \vdash S : \text{ok}$) if shared objects are allocated.

$$S \subseteq \text{dom}(H)$$

Definition 10. A lock map is well-formed with respect to a heap and shared set $(H, S \vdash L : \text{ok})$ if locked objects are shared.

$$\text{dom}(L) \subseteq S$$

Definition 11. A program state

$$\sigma = (H, L, S, T)$$

is well-formed ($\text{wf}(\sigma)$) if the following conditions hold:

- The access sets and the free set partition the heap.

$$\left(\bigsqcup_{(\cdot, A, \cdot) \in T} \{A\} \right) \uplus \{S \setminus \text{dom}(L)\} \ll \text{dom}(H)$$

- The heap, the shared set, and the lock map are well-formed.

$$\vdash H : \text{ok} \wedge H \vdash S : \text{ok} \wedge H, S \vdash L : \text{ok}$$

- The continuations in each thread's call stack contain only references to allocated objects and do not contain any free variables.

$$\forall (\cdot, \cdot, F) \in T \bullet \forall \bar{s} \in F \bullet \\ (\text{objectRefs}(\bar{s}) \subseteq \text{dom}(H) \wedge \text{free}(\bar{s}) = \emptyset)$$

Notice that well-formedness of a program state implies that access sets are disjoint and that accessible objects are allocated.

Theorem 1. In a legal program, the small step relation \rightarrow preserves well-formedness.

$$\text{program.legal} \Rightarrow (\forall \sigma_1, \sigma_2 \bullet (\text{wf}(\sigma_1) \wedge \sigma_1 \rightarrow \sigma_2) \Rightarrow \text{wf}(\sigma_2))$$

Proof. By case analysis on the step from σ_1 to σ_2 . We consider cases SHARE, SYNCHRONIZED, and UNLOCK.

- **Case SHARE.** By legality, we have that the object being shared is in the access set but not in the shared set. The step adds it to the shared set (and therefore the free set since unshared objects are not locked) and removes it from the access set. It follows that the partition is maintained.
- **Case SYNCHRONIZED.** Assume that the object being locked is o . In σ_1 , o is shared and not locked, and therefore it is part of σ_1 's free set. Since in a well-formed state the free set and the access sets partition the heap, o is not in any thread's access set. Adding o to a single access set and removing it from the free set (by adding an entry for o to the lock map) maintains the proper partitioning of the heap. Because locking an object modifies neither the heap nor the shared set and both are well-formed in the pre-state, they are well-formed in the post-state. Adding an entry for o (a shared object) to σ_1 's well-formed lock set, preserves the fact that the lock set only contains shared objects. Finally, the continuations in σ_2 contain neither free variables nor unallocated objects, because σ_1 does not contain any and because locking did not introduce any.

- **Case UNLOCK.** By legality, the object being unlocked is in the access set and is locked by the current thread. The step removes it from the lock map (thus adding it to the free set, since locked objects are shared) and from the thread's access set. It follows that the partition is maintained.

□

Theorem 2. *States reached by legal programs are well-formed.*

Proof. By induction on the number of execution steps: the initial state is well-formed, and the induction step holds, as per Theorem 1. □

The notion of *data race* is defined in JLS3 in terms of the *happens-before* relation.

Definition 12 (Happens-Before). *Consider an execution of a legal program. The happens-before relation on the execution is a partial order among the steps of the execution. Specifically, it is the smallest transitive relation such that*

- *Each step performed by a thread t happens-before each subsequent step performed by t .*
- *Each UNLOCK step on an object o happens-before each subsequent SYNCHRONIZED step on o .*
- *Each NEWTHREAD step that creates a thread t happens-before each operation performed by t .*

Definition 13 (Data Race). *Consider an execution of a legal program. A pair of steps of the execution constitutes a data race if one is a WRITE of a field $o.f$ and the other is a READ or WRITE of $o.f$ and the steps are not ordered by the happens-before relation.*

The following lemma states that in legal programs, no ordering constraints exist on execution steps beyond those imposed by the synchronization constructs (i.e., thread creation and **synchronized** blocks).

Lemma 1. *In an execution of a legal program π , if two consecutive steps are not ordered by happens-before, then swapping them results again in an execution of π .*

Proof. By case analysis on the steps. We detail a few cases.

- The steps are not accesses of the same field, since this would mean access sets are not disjoint, and by Theorem 2 we have that program states are well-formed.
- A NEW step can be moved to the right. The other step does not access the newly created object since the NEW step does not modify the thread state of the other thread and well-formedness of the latter implies it does not mention unallocated objects.

□

We are now ready to prove the main theorem of this section.

Theorem 3. *Executions of legal programs do not contain data races.*

Proof. By contradiction. Consider a legal program π such that at least one of its executions contains a data race on a field $o.f$. Of all the data races in all the executions of π , pick one where the number of steps that intervene between the steps that constitute the data race is minimal. Let those steps be S_i and S_{i+1+n} .

Assume first that $n = 0$. In state σ_{i-1} (the pre-state of step S_i) o is in the access set of the thread t_i that performs S_i . Since a field access does not modify any access sets, this is still true in state σ_i . However, in this state, it is also true that o is in the access set of thread t_{i+1} that performs S_{i+1} . This contradicts Theorem 2.

Assume now that $n > 0$. Call S_j the last step preceding S_{i+1+n} that does not happen before S_{i+1+n} . Since S_i precedes S_{i+1+n} and does not happen before S_{i+1+n} , such a step exists, and either $i = j$ or $i < j$. We consider two cases:

- **Case $i = j$.** By repeated application of Lemma 1, step S_i can be moved directly before S_{i+1+n} , which results in a data race without intervening steps, contradicting the minimality assumption.
- **Case $i < j$.** By repeated application of Lemma 1, step S_j can be moved after S_{i+1+n} , which results in a data race with $n - 1$ intervening steps, contradicting the minimality assumption.

□

1.4 Non-interference

In this subsection, we introduce the notion of a *non-interfering state change* and we prove that in executions of legal programs, with respect to the access set of one thread, steps of other threads are non-interfering state changes.

Definition 14. *Two states are related by a non-interfering state change with respect to a given access set if*

- *all objects that are allocated in the first state are also allocated in the second state,*
- *all objects that are shared in the first state are also shared in the second state, and*
- *if an object is in the access set, then its state in the heap is unchanged and if additionally the object is unshared in the first state, then it is unshared in the second state.*

Formally:

$$\begin{array}{c} (H, S) \overset{A}{\rightsquigarrow} (H', S') \\ \updownarrow \\ \text{dom}(H) \subseteq \text{dom}(H') \wedge S \subseteq S' \wedge H|_A = H'|_A \wedge S' \cap A = S \cap A \end{array}$$

The following theorem states a key property of the programming model. Note: we write $\sigma \xrightarrow{\text{tid}} \sigma'$ to denote that program states σ and σ' are related by an execution step performed by thread tid .

Theorem 4 (Thread Isolation). *In an execution of a legal program, with respect to the access set of one thread, a sequence of steps of other threads constitutes a non-interfering state change.*

$$\begin{array}{c}
\text{program_legal} \wedge \text{initial}(\sigma^0) \wedge \sigma^0 \xrightarrow{*} \sigma_0 \xrightarrow{\text{tid}_1} \sigma_1 \cdots \xrightarrow{\text{tid}_n} \sigma_n \\
\wedge \\
(\forall i \in \{0, \dots, n\} \bullet \sigma_i = (\mathbf{H}_i, \mathbf{L}_i, \mathbf{S}_i, \mathbf{T}_i \triangleleft (\text{tid}, \mathbf{A}_i, \mathbf{F}_i))) \wedge \text{tid} \notin \{\text{tid}_1, \dots, \text{tid}_n\} \\
\Downarrow \\
(\mathbf{H}_0, \mathbf{S}_0) \overset{\mathbf{A}_0}{\rightsquigarrow} (\mathbf{H}_n, \mathbf{S}_n) \wedge \mathbf{A}_n = \mathbf{A}_0
\end{array}$$

Proof. Since the program is legal, by Theorem 2 we have that all program states reached are legal and well-formed. We prove the theorem by induction on n . The base case $n = 0$ is trivial. Assume $n > 0$. By induction we have

$$(\mathbf{H}_0, \mathbf{S}_0) \overset{\mathbf{A}_0}{\rightsquigarrow} (\mathbf{H}_{n-1}, \mathbf{S}_{n-1}) \wedge \mathbf{A}_{n-1} = \mathbf{A}_0$$

We perform case analysis on the rule used to derive the last step. We consider case WRITE.

Suppose the step's pre-state is well-formed and that the step writes value v to field f of object o . No new objects are allocated when assigning to fields, and therefore the domain of the old and the new heap are equal. The old and the new heap differ only at a single location, namely (o, f) . From legality it follows that o is an element of thread tid_n 's access set, and by well-formedness it is not an element of \mathbf{A}_0 . Finally, updating a field does not modify the shared set. \square

2 Static verification

The previous section proved that legal programs are data-race-free. However, legality of a program cannot be checked automatically. In this section, we define the notion of *valid* programs, which is a condition that is modular and that is suitable for submission to an automatic theorem prover, and we show that valid programs are legal. It follows that valid programs are data-race-free and that they perform no null dereferences or ill-typed operations.

The validity notion is based on provability of *verification conditions* in the *verification logic*, i.e., the logic used to interpret the verification conditions.

Before we define and prove the validity notion, we establish the verification logic and we discuss the modular verification approach.

2.1 Verification logic

We target multi-sorted first-order predicate logic with equality. 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* from the *signature* with *arity* n . A *formula* ϕ is an *equality* $t_1 = t_2$, an *inequality* $t_1 \neq t_2$, a *literal true or false*, an *atom* $P(t_1, \dots, t_n)$ where P is a predicate symbol from the signature with arity n , a *propositional formula* using the connectives \wedge , \vee , \Rightarrow , and \neg , or a *quantification* $(\forall y \bullet \phi)$.

The sorts are the following:

- The sort of object references
- The sort of program values (the object references and the null value)

- The sort of finite sets of object references
- The sort of field names
- The sort of class names
- The sort of interface names
- The sort of *object states*, i.e., finite partial functions from field names to program values
- The sort of *heaps*, i.e., finite partial functions from object references to object states

Note:

- All sorts are countably infinite.
- We leave the sorts of quantifications, function symbols, and predicate symbols implicit when they are clear from the context.

We use the following signature, for a given program:

Predicate symbol	Syntax	Notes
$\text{in}(t_1, t_2)$	$t_1 \in t_2$	
$\text{prec}(t_1, t_2)$	$t_1 \prec t_2$	subtype relation

Function symbol	Syntax	Notes
emptyset	\emptyset	empty set, empty function
$\text{insert}(t_1, t_2)$	$t_1 \cup \{t_2\}$	
$\text{insertnonnull}(t_1, t_2)$	$t_1 \cup \{t_2\}_{-\text{null}}$	
$\text{remove}(t_1, t_2)$	$t_1 \setminus \{t_2\}$	
$\text{intersect}(t_1, t_2)$	$t_1 \cap t_2$	
$\text{setminus}(t_1, t_2)$	$t_1 - t_2$ or $t_1 \setminus t_2$	
$\text{apply}(t_1, t_2)$	$t_1(t_2)$	
$\text{update}(t_1, t_2, t_3)$	$t_1[t_2 \mapsto t_3]$	function update
$\text{dom}(t)$	$\text{dom}(t)$	function domain
null	null	
$\text{classof}(t)$	$\text{classof}(t)$	class of an object
$\text{asvalue}(t)$	t	implicit widening conversion
$\text{asobjref}(t)$	t	implicit narrowing conversion

The widening and narrowing conversions are inserted implicitly to convert between program values and object references. Also, some of the function symbols are overloaded. Specifically, implicitly there is a separate emptyset symbol for each set or function sort, and there are separate apply , update , and dom symbols for each function sort.

Additionally, the signature contains a nullary function symbol τ for each class or interface τ declared by the program, and a nullary function symbol f for each field f declared by the program.

Note: we also use the following abbreviations:

Abbreviation	Meaning
$t_1(t_2, t_3)$	$t_1(t_2)(t_3)$
$t_1[(t_2, t_3) \mapsto t_4]$	$t_1[t_2 \mapsto (t_1(t_2)[t_3 \mapsto t_4])]$
$t_1 \subseteq t_2$	$(\forall y \bullet y \in t_1 \Rightarrow y \in t_2)$
$t_2 _{t_1} = t_3 _{t_1}$	$(\forall y \bullet y \in t_1 \Rightarrow t_2(y) = t_3(y))$
$t_1 \preceq t_2$	$t_1 \prec t_2 \vee t_1 = t_2$

where y is a fresh logical variable.

Throughout, we interpret the logic according to an interpretation \mathcal{I} of the signature, which is as expected. Corner cases are as follows. $\mathcal{I}(\text{asobjref})(\mathcal{I}(\text{null}))$ yields some (fixed) object reference. If e_2 is not in the domain of e_1 , then $\mathcal{I}(\text{apply})(e_1, e_2)$ yields some (fixed) element of the range sort. $\mathcal{I}(\text{prec})(C, I)$ holds if the program declares a class C that mentions an interface I in its **implements** clause. Function $\mathcal{I}(\text{insertnonnull})(S, v)$ returns S if v is null, and $S \cup \{v\}$ otherwise.

Let Σ be an (incomplete) axiomatization of \mathcal{I} . By the soundness of first-order logic, it follows that if a formula ϕ is provable from Σ , then ϕ is true under \mathcal{I} :

$$\text{if } \Sigma \vdash \phi \text{ then } \mathcal{I} \models \phi$$

Note:

- A multi-sorted first-order logic where all sorts are countably infinite can be reduced to a one-sorted first-order logic by having a universe of natural numbers and mapping it to the various sorts as appropriate in the interpretation. This allows us to use theorem provers for one-sorted logic, such as Simplify.
- We will sometimes use *partially interpreted formulae*, i.e., formulae where elements of the universe appear as terms. Also, we will sometimes use a verification logic formula directly as a meta-logic formula, assuming the interpretation \mathcal{I} and interpreting the free logical variables as the corresponding meta-logical variables.

2.2 Thread-relevant state and state predicates

Definition 15. *In a given program state σ , the thread-relevant state (H, L_t, S, A, B) of a certain thread consists of the heap, the objects locked by the thread, the shared set, the thread's access set, and the bound set B of block-bound objects.*

Definition 16. *A thread-relevant state (H, L_t, S, A) is well-formed ($\vdash (H, L_t, S, A) : \text{ok}$) if the heap is well-formed, the shared set is well-formed with respect to the heap, the access set is a subset of the heap's domain, the lock set is a subset of the shared set, the bound set is a subset of the heap's domain, non-null field values are shared, and if an object's `inv` field is `true`, then its invariant holds.*

$$\begin{aligned}
& \vdash (H, L_t, S, A) : \text{ok} \\
& \quad \updownarrow \\
& \vdash H : \text{ok} \wedge H \vdash S : \text{ok} \wedge L_t \subseteq S \wedge A \subseteq \text{dom}(H) \wedge B \subseteq \text{dom}(H) \\
& \quad \wedge \\
& \quad (\forall o \in \text{dom}(H), f \in \text{dom}(H(o)) \bullet H(o, f) = \text{null} \vee H(o, f) \in S) \\
& \quad \wedge \\
& \quad (\forall o \in \text{dom}(H) \bullet H(o, \text{inv}) = \text{true} \Rightarrow \text{invariant}(\text{classof}(o))[o/\text{this}])
\end{aligned}$$

A continuation, as it appears in a program text, may contain free program variables. Consequently, the corresponding continuation verification condition contains these program variables as free logical variables.

Definition 17. A program variable $z \in \text{Var}$ is either **this**, **result**, or a method parameter or local variable $x \in \mathcal{X}$.

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

A continuation verification condition is a *state predicate*. A state predicate may be a *two-state predicate* or a *one-state predicate*. A two-state predicate may refer to the current thread-relevant state (using free variables H , L_t , S , and A) and to the *old state* (using free variables H^{old} , L_t^{old} , S^{old} , and A^{old}). A one-state predicate may refer only to the current state (using free variables H , L_t , S , and A). Since method postconditions are two-state predicates, so are continuation verification conditions.

Definition 18. A two-state predicate (or state predicate for short) Q is a formula of the verification logic, whose free logical variables are the current and old thread-relevant state variables and program variables:

$$\text{free}(Q) \subseteq \{H, L_t, S, A, B, H^{\text{old}}, L_t^{\text{old}}, S^{\text{old}}, A^{\text{old}}\} \cup \text{Var}$$

A state predicate Q is called a one-state predicate if the old thread-relevant state variables do not appear free in it.

$$\text{free}(Q) \subseteq \{H, L_t, S, A\} \cup \text{Var}$$

A state predicate is *program-closed* if no program variables appear free in it.

We will use the notation $\mathfrak{J}, H, L_t, S, A, V \models Q$ to denote the truth of a state predicate in a particular thread-relevant state and under a particular valuation V that maps program variables to program values.

Definition 19. A state predicate Q is *local* if it is preserved by a non-interfering state change.

$$\begin{aligned} \text{local}(Q) \equiv & (\forall \bar{v}, H^{\text{old}}, L_t^{\text{old}}, S^{\text{old}}, A^{\text{old}}, H, L_t, S, A, H', S' \bullet \\ & Q[\bar{v}/\bar{g}] \wedge (H, S) \stackrel{A}{\rightsquigarrow} (H', S') \Rightarrow Q[\bar{v}/\bar{g}, H'/H, S'/S]) \end{aligned}$$

where \bar{g} are the program variables that appear free in Q .

2.3 Modular verification

To determine validity of a method, one could take the entire program into account at once. However, this approach is not modular, and hence does not scale to large programs. Instead, we propose a modular approach where the validity of a method does not depend on the entire program, but only on its body, its *method contract*, and the contracts of its callees.

A method contract is of the form

requires P ; **ensures** Q ;

where P is a state predicate and Q is a two-state predicate. Specifically, the free variables allowed in P are H, L_t, S , and A , as well as **this** and the method's parameters. The free variables allowed in Q are $H, L_t, S, A, H^{\text{old}}, L_t^{\text{old}}, S^{\text{old}}$, and A^{old} , as well as **this**, the method's parameters, and **result**. P is called the method's precondition and Q is called the method's postcondition.

To verify whether a method complies with the model and respects its method contract and those of its callees, one may only assume that the method's precondition holds and that the program state is well-formed when the method is called. Furthermore, note that methods cannot query the access set in the sense that branching on whether an object is accessible or not is impossible. From this we can derive a framing property: a method call does not modify or share an object o that is allocated and accessible before the call, if the precondition does not require o to be accessible. Thanks to this framing property, we do not need explicit modifies clauses in method contracts.

Consider a formula φ in the verification logic such that $\text{free}(\varphi) \subseteq \{A\}$. The *required access set* $\mathcal{A}(\varphi)$ of φ is defined as

$$\{o \in \mathcal{O} \mid \forall A \subseteq \mathcal{O} \bullet \varphi \Rightarrow o \in A\}$$

That is, when interpreting φ as a set of access sets, we have

$$\mathcal{A}(\varphi) = \bigcap \varphi$$

In our verification approach, for a given method precondition P , we compute a first-order expression for $\mathcal{A}(P)$ syntactically. This is possible if the syntax of preconditions is restricted sufficiently.

We require method preconditions and postconditions to be local.

We impose the additional restriction that fields can only hold shared objects. We verify that any field write stores a shared object; as a result, we may assume that the objects read from fields are shared. This restriction simplifies the formal development, but it can easily be relieved, for example by introducing a **shared** modifier, which indicates that a certain field can only hold shared objects, or by introducing invariants (see Section ??). Notice that the correctness of the example of Figure 2 depends on this feature.

2.4 Valid programs

In this subsection, we define which *verification conditions* are generated for a given program. A verification condition is a closed formula in the verification logic. A program is *valid* if its verification conditions are provable from the verification logic's theory Σ .

We define program verification conditions in terms of *continuation verification conditions*. A continuation verification condition $\text{vc}(\bar{s}, Q)$ is a formula of the verification logic that expresses conditions under which a continuation \bar{s} (i.e., a list of statements to be executed next) executes correctly and satisfies postcondition Q when started in a given program state. A continuation verification condition depends only on the state variables relevant to the thread that executes the continuation. These state variables appear as free logical variables in the formula.

Figure 6 defines the predicate transformer vc , which maps a continuation \bar{s} and a state predicate Q to the continuation verification condition for \bar{s} and Q .

As noted above, $\text{vc}(\bar{s}, Q)$ is a sufficient condition such that any thread in a state satisfying $\text{vc}(\bar{s}, Q)$ ends up in a state satisfying Q after executing the continuation \bar{s} . Note: applications of the predicate transformer have higher priority than substitutions; in other words, to compute a verification condition, first apply all verification condition rules exhaustively and then apply the substitutions. $\text{pre}(m)$ and $\text{post}(m)$ denote the precondition and postcondition, respectively, declared by method m .

The intuition underlying the verification condition rules is as follows. The verification condition for VC-IF is a standard weakest precondition. Since an **assert** statement blocks forever if its condition is false, the statement's continuation is verified under the assumption that the condition is true. For a field access (VC-READ or VC-WRITE), the target of the access must be non-null and accessible. A newly created object (VC-NEW) was previously unallocated and is of the correct class. In order to share (VC-SHARE) a value, the value should be non-null, accessible and unshared. Locking (VC-SYNCHRONIZED) is disallowed if the object is unshared or already locked by the current thread. Moreover, since between synchronized statements other threads may modify shared objects, we should assume nothing about the fields of the newly locked object. The only property we may assume is that the old state and the new state are related by a non-interfering state change with respect to the thread's access set. Unlocking (VC-UNLOCK) is allowed only for locked and accessible objects. When invoking a method (VC-CALL), the target should not be null, the callee's precondition should hold and when returning (VC-RECEIVE) the postcondition should hold. When a method call returns, we make some assumptions about the post-state. First of all, as per the method framing approach, we may assume that the post-state is related to the pre-state by a non-interfering state change with respect to the pre-state access set minus the callee's required access set. Secondly, we may assume the post-state is well-formed and satisfies the callee's postcondition. Finally, we may assume the return value is allocated. Starting a new thread via **start** (VC-NEWTREAD) requires the target object to be accessible and non-null. Accessibility of the target object is transferred from the current thread to the new thread.

An important property of continuation verification conditions is that they are *local*.

Theorem 5. *If a state predicate Q is local, then the verification condition of a continuation \bar{s} with respect to Q is local as well.*

$$\text{local}(Q) \Rightarrow \text{local}(\text{vc}(\bar{s}, Q))$$

Proof. By induction on \bar{s} .

- **Case $\bar{s} = \text{synchronized}(v) \{ \bar{s}'' \} \bar{s}'$.**

1. We may assume that \bar{s} is valid (i.e., $\text{vc}(\bar{s}, Q)$ holds) in a state (H, L_t, S, A) .
2. It follows by VC-SYNCHRONIZED that the continuation \bar{s}'' **unlock** v ; \bar{s}' of \bar{s} is valid in any state $(H', L_t \cup \{v\}, S', A \cup \{v\})$ where $(H, S) \stackrel{A}{\rightsquigarrow} (H', S')$.
3. We need to prove that \bar{s} is valid in any state (H'', L_t, S'', A) where $(H, S) \stackrel{A}{\rightsquigarrow} (H'', S'')$.

$$\begin{aligned}
& \text{vc}(\mathbf{if} (v_1 = v_2) \{ \bar{s}_1 \} \mathbf{else} \{ \bar{s}_2 \} \bar{s}, \phi, \bar{b}, Q) \equiv & \text{[VC-IF]} \\
& \quad (v_1 = v_2 \Rightarrow \text{vc}(\bar{s}_1 \bar{s}, \phi, \bar{b}, Q)) \wedge (v_1 \neq v_2 \Rightarrow \text{vc}(\bar{s}_2 \bar{s}, \phi, \bar{b}, Q)) \\
& \text{vc}(\mathbf{assert} \ v \ \mathbf{instanceof} \ \tau; \bar{s}, \phi, \bar{b}, Q) \equiv & \text{[VC-ASSERT]} \\
& \quad (v \neq \text{null} \wedge \text{classof}(v) \preceq \tau) \Rightarrow \text{vc}(\bar{s}, \phi, \bar{b}, Q) \\
& \text{vc}(x := v.f; \bar{s}, \phi, \bar{b}, Q) \equiv & \text{[VC-READ]} \\
& \quad v \neq \text{null} \wedge \text{classof}(v) = \text{declaringClass}(f) \wedge v \in A \wedge (\phi \wedge \text{vc}(\bar{s}, \phi, \bar{b}, Q))[\text{H}(v, f)/x] \\
& \text{vc}(v_1.f := v_2; \bar{s}, \bar{b}, Q) \equiv & \text{[VC-WRITE]} \\
& \quad v_1 \neq \text{null} \wedge \text{classof}(v_1) = \text{declaringClass}(f) \wedge v_1 \in A \wedge (v_2 = \text{null} \vee v_2 \in S) \\
& \quad \wedge \text{H}(v_1, \text{inv}) = \text{false} \wedge f \neq \text{inv} \wedge (\phi \wedge \text{vc}(\bar{s}, \phi, \bar{b}, Q))[\text{H}[(v_1, f) \mapsto v_2]/\text{H}] \\
& \text{vc}(x := \mathbf{new} \ C; \bar{s}, \phi, \bar{b}, Q) \equiv & \text{[VC-NEW]} \\
& \quad \forall o \bullet o \notin \text{dom}(\text{H}) \wedge \text{classof}(o) = C \Rightarrow \\
& \quad (\phi \wedge \text{vc}(\bar{s}, \phi, \bar{b}, Q))[o/x, \text{H}[o \mapsto \emptyset[f_1 \mapsto \text{null}] \cdots [f_n \mapsto \text{null}][\text{inv} \mapsto \text{false}]/\text{H}, (A \cup \{o\})/A] \\
& \quad \text{where fields}(C) = \{f_1, \dots, f_n\} \\
& \text{vc}(\mathbf{share} \ v; \bar{s}, \phi, \bar{b}, Q) \equiv & \text{[VC-SHARE]} \\
& \quad v \neq \text{null} \wedge v \in A \wedge v \notin S \wedge v \notin B \wedge (\phi \wedge \text{vc}(\bar{s}, \phi, \bar{b}, Q))[(A \setminus \{v\})/A, (S \cup \{v\})/S] \\
& \text{vc}(\mathbf{synchronized} \ (v) \{ \bar{s}' \} \bar{s}, \phi, \bar{b}, Q) \equiv & \text{[VC-SYNCHRONIZED]} \\
& \quad v \neq \text{null} \wedge v \in S \wedge v \notin L_t \\
& \quad \wedge (\forall H', S' \bullet \\
& \quad \quad ((H, S) \overset{A}{\rightsquigarrow} (H', S') \wedge \vdash (H', L_t, S', A) : \text{ok}) \\
& \quad \Rightarrow \\
& \quad (\phi \wedge \text{vc}(\bar{s}', \phi, (\{v\}, \mathbf{unlock} \ v; \bar{s}, \phi) \cdot \bar{b}, Q))[\text{H}'/\text{H}, (A \cup \{v\})/A, (L_t \cup \{v\})/L_t, S'/S, B \cup \{v\}/B]) \\
& \text{vc}(\epsilon, \phi, (B_\Delta, \mathbf{unlock} \ o; \bar{s}, \phi) \cdot \bar{b}, Q) \equiv & \text{[VC-UNLOCK]} \\
& \quad (\phi \wedge \text{vc}(\bar{s}, \phi, \bar{b}, Q))[(A \setminus \{o\})/A, (L_t \setminus \{o\})/L_t, B \setminus \{o\}/B] \\
& \text{vc}(x := v.m(\bar{v}); \bar{s}, \phi, \bar{b}, Q) \equiv & \text{[VC-CALL]} \\
& \quad v \neq \text{null} \wedge \text{classof}(v) \preceq \text{declaringType}(m) \wedge \text{pre}(m)[v/\mathbf{this}, \bar{v}/\bar{x}] \\
& \quad \wedge \text{vc}(x := \mathbf{receive} \ [(H, L_t, S, A, B)] \ v.m(\bar{v}); \bar{s}, \phi, \bar{b}, Q) \\
& \text{vc}(\mathbf{start} \ v.m(); \bar{s}, \phi, \bar{b}, Q) \equiv & \text{[VC-NEWTREAD]} \\
& \quad v \neq \text{null} \wedge \text{classof}(v) \preceq \text{declaringType}(m) \wedge v \in A \wedge v \notin B \wedge (\phi \wedge \text{vc}(\bar{s}, \bar{b}, Q))[(A \setminus \{v\})/A] \\
& \text{vc}(\mathbf{return} \ v; \phi, \epsilon, Q) \equiv & \text{[VC-RETURN]} \\
& \quad Q[v/\mathbf{result}]
\end{aligned}$$

Figure 6: Continuation verification conditions (*Part 1 of 3*)

$$\begin{aligned}
\text{vc}(x := \text{receive } [(H^{\text{pre}}, L_t^{\text{pre}}, S^{\text{pre}}, A^{\text{pre}}, B^{\text{pre}})] o.m(\bar{v}); \bar{s}, \phi, \bar{b}, Q) &\equiv \quad [\text{VC-RECEIVE}] \\
\forall H', S', A', v_r \bullet & \\
((H, S) \overset{A_{\text{caller}}}{\rightsquigarrow} (H', S') \wedge (\vdash (H', L_t, S', A') : \text{ok}) & \\
\wedge A' \cap A_{\text{caller}} = A \cap A_{\text{caller}} & \\
\Rightarrow & \\
(\phi \wedge & \\
(Q'[v_r/\text{result}, H^{\text{pre}}/H^{\text{old}}, L_t^{\text{pre}}/L_t^{\text{old}}, S^{\text{pre}}/S^{\text{old}}, A^{\text{pre}}/A^{\text{old}}] & \\
\wedge (v_r = \text{null} \vee v_r \in \text{dom}(H)) & \\
\Rightarrow & \\
(\phi \wedge \text{vc}(\bar{s}, \phi, \bar{b}, Q))[v_r/x]) & [H'/H, S'/S, A'/A, L_t^{\text{pre}}/L_t, B^{\text{pre}}/B] \\
\text{where } P' = \text{pre}(m)[o/\text{this}, \bar{v}/\bar{x}] \text{ and } Q' = \text{post}(m)[o/\text{this}, \bar{v}/\bar{x}] & \\
\text{and } A_{\text{caller}} = A^{\text{pre}} - \mathcal{A}(P'[H^{\text{pre}}/H, L_t^{\text{pre}}/L_t, S^{\text{pre}}/S]) &
\end{aligned}$$

$$\begin{aligned}
\text{vc}(\text{par } v_1.m_1() \parallel v_2.m_2(); \bar{s}, \phi, \bar{b}, Q) &\equiv \quad [\text{VC-PAR}] \\
v_1 \neq \text{null} \wedge \text{classof}(v_1) \preceq \text{declaringType}(m_1) \wedge \text{pre}(m_1)[v_1/\text{this}] & \\
\wedge v_2 \neq \text{null} \wedge \text{classof}(v_2) \preceq \text{declaringType}(m_2) \wedge \text{pre}(m_2)[v_2/\text{this}] & \\
\wedge \mathcal{A}(\text{pre}(m_1)[v_1/\text{this}]) \cap \mathcal{A}(\text{pre}(m_2)[v_2/\text{this}]) = \emptyset & \\
\wedge (\forall \text{tid}' \bullet \text{vc}(\text{receive_par } [(H, L_t, S, A, B), \text{tid}'] v_1.m_1() \parallel v_2.m_2(); \bar{s}, \phi, \bar{b}, Q)) &
\end{aligned}$$

$$\begin{aligned}
\text{vc}(\text{receive_par } [(H, L_t, S, A, B), \text{tid}'] o_1.m_1() \parallel o_2.m_2(); \bar{s}, \phi, \bar{b}, Q) &\equiv \quad [\text{VC-RECEIVE-PAR}] \\
\forall H', S', A' \bullet & \\
((H, S) \overset{A_{\text{caller}}}{\rightsquigarrow} (H', S') \wedge (\vdash (H', L_t, S', A') : \text{ok}) & \\
\wedge A' \cap A_{\text{caller}} = A \cap A_{\text{caller}} & \\
\Rightarrow & \\
(\phi \wedge & \\
(Q_1 \wedge Q_2 & \\
\Rightarrow & \\
\text{vc}(\bar{s}, \phi, \bar{b}, Q))[H'/H, S'/S, A'/A, L_t^{\text{pre}}/L_t, B^{\text{pre}}/B] & \\
\text{where } P_1 = \text{pre}(m_1)[o_1/\text{this}, H^{\text{pre}}/H, L_t^{\text{pre}}/L_t, S^{\text{pre}}/S] & \\
\text{and } Q_1 = \text{post}(m_1)[o_1/\text{this}, & \\
L_t^{\text{pre}}/L_t, H^{\text{pre}}/H^{\text{old}}, S^{\text{pre}}/S^{\text{old}}, A^{\text{pre}}/A^{\text{old}}] & \\
\text{and } P_2 = \text{pre}(m_2)[o_2/\text{this}, H^{\text{pre}}/H, L_t^{\text{pre}}/L_t, S^{\text{pre}}/S] & \\
\text{and } Q_2 = \text{post}(m_2)[o_2/\text{this}, & \\
L_t^{\text{pre}}/L_t, H^{\text{pre}}/H^{\text{old}}, S^{\text{pre}}/S^{\text{old}}, A^{\text{pre}}/A^{\text{old}}] & \\
\text{and } A_{\text{caller}} = A^{\text{pre}} - \mathcal{A}(P_1) - \mathcal{A}(P_2) &
\end{aligned}$$

$$\begin{aligned}
\text{vc}(\text{pack}_C v; \bar{s}, \phi, \bar{b}, Q) &\equiv \quad [\text{VC-PACK}] \\
v \neq \text{null} \wedge \text{classof}(v) = C \wedge \text{invariant}(C)[v/\text{this}] & \\
\wedge (\phi \wedge \text{vc}(\bar{s}, \phi, \bar{b}, Q))[H[(v, \text{inv}) \mapsto \text{true}]/H] &
\end{aligned}$$

$$\begin{aligned}
\text{vc}(\text{unpack}_C v; \bar{s}, \phi, \bar{b}, Q) &\equiv \quad [\text{VC-UNPACK}] \\
v \neq \text{null} \wedge \text{classof}(v) = C & \\
\wedge (\phi \wedge \text{vc}(\bar{s}, \phi, \bar{b}, Q))[H[(v, \text{inv}) \mapsto \text{false}]/H] &
\end{aligned}$$

Figure 6: Continuation verification conditions (*Part 2 of 3*)

$$\begin{aligned}
\text{vc}(\text{throw}; \bar{s}, \phi, \bar{b}, Q) &\equiv \text{true} && \text{[VC-THROW]} \\
\text{vc}(\text{try binding } v_1, \dots, v_n; \text{invariant } \phi'; \{ \bar{s}_1 \} \text{ catch } \{ \bar{s}_2 \} \bar{s}, \phi, \bar{b}, Q) &\equiv && \text{[VC-TRY]} \\
&v_1 \neq \text{null} \wedge \dots \wedge v_n \neq \text{null} \\
&\wedge v_1 \in \mathbf{A} \wedge \dots \wedge v_n \in \mathbf{A} \\
&\wedge (\phi \wedge \phi' \wedge \text{vc}(\bar{s}_1, \phi \wedge \phi', (\{v_1, \dots, v_n\}, \text{catch } [(H, L_t, S, A, B), \{\bar{x}\}, \phi'] \{ \bar{s}_2 \} \bar{s}, \phi) \cdot \bar{b}, Q)[\\
&\quad B \cup \{v_1, \dots, v_n\}/B]) \\
&\wedge \text{vpc}((\{v_1, \dots, v_n\}, \text{catch } [(H, L_t, S, A, B), \{\bar{x}\}, \phi'] \{ \bar{s}_2 \} \bar{s}, \phi) \cdot \bar{b}, Q) \\
&\text{where } \bar{x} \text{ are the target variables of the assignments in } \bar{s}_1 \\
\text{vc}(\epsilon, \phi, (B_\Delta, \text{catch } [(H^{\text{pre}}, L_t^{\text{pre}}, S^{\text{pre}}, A^{\text{pre}}, B^{\text{pre}}), \{\bar{x}\}, \phi''] \{ \bar{s} \} \bar{s}', \phi') \cdot \bar{b}, Q) &\equiv && \text{[VC-CATCH]} \\
&(\phi' \wedge \text{vc}(\bar{s}', \phi', \bar{b}, Q))[B^{\text{pre}}/B] \\
\text{vpc}(\epsilon, (B_\Delta, &&& \text{[VPC-CATCH]} \\
\text{catch } [(H^{\text{pre}}, L_t^{\text{pre}}, S^{\text{pre}}, A^{\text{pre}}, B^{\text{pre}}), \{\bar{x}\}, \phi] \{ \bar{s} \} \bar{s}', \phi') \cdot \bar{b}, Q) &\equiv \\
(\forall H, S, A, \bar{v} \bullet &&& \\
((\vdash (H, L_t, S, A, B^{\text{pre}}) : \text{ok} \wedge B^{\text{pre}} \cup B_\Delta \subseteq A \wedge \phi \wedge \phi') &&& \\
\Rightarrow (\phi' \wedge \text{vc}(\bar{s} \bar{s}', \phi', \bar{b}, Q))[B^{\text{pre}}/B])[L_t^{\text{pre}}/L_t, \bar{v}/\bar{x}]) \\
\text{vpc}(\epsilon, (B_\Delta, \text{unlock } o; \bar{s}, \phi') \cdot \bar{b}, Q) &\equiv \text{true} && \text{[VPC-UNLOCK]}
\end{aligned}$$

Figure 6: Continuation verification conditions (*Part 3 of 3*)

4. This requires that we prove that the continuation $\bar{s}'' \text{ unlock } v; \bar{s}'$ of \bar{s} is valid in any state $(H''', L_t \cup \{v\}, S''', A \cup \{v\})$ where $(H'', S'') \stackrel{\Delta}{\rightsquigarrow} (H''', S''')$.
5. It is easy to see that the non-interfering state change relation is transitive; therefore, from the assumptions in points 3 and 4 we have $(H, S) \stackrel{\Delta}{\rightsquigarrow} (H''', S''')$. By instantiating the rule in point 2, we obtain the goal in point 4.

- **Case receive** is analogous to case **synchronized**.
- The other cases are easy.

□

We are now ready to define *program validity*.

Definition 20. *A well-formed program is valid if all of the following hold:*

- each method precondition, method postcondition, and block invariant ϕ is local

$$\Sigma \vdash \text{local}(\phi)$$

- each method $m(\bar{x})$ **requires** P ; **ensures** Q ; $\{ \bar{s} \}$ in each class C is valid, i.e., it is provable from the verification logic that the precondition implies

validity of the method's body with respect to the postcondition.

$$\begin{array}{c} \Sigma \vdash \\ \forall o, \bar{v}, H, L_t, S, A, B \bullet \\ (\vdash (H, L_t, S, A, B) : \text{ok} \wedge \text{classof}(o) = C \wedge P[o/\mathbf{this}, \bar{v}/\bar{x}]) \\ \downarrow \\ \text{vc}(\bar{s}, Q)[o/\mathbf{this}, \bar{v}/\bar{x}, H/H^{\text{old}}, L_t/L_t^{\text{old}}, S/S^{\text{old}}, A/A^{\text{old}}] \end{array}$$

- the main routine \bar{s} is valid:

$$\Sigma \vdash \text{vc}(\bar{s}, \mathbf{true})[\emptyset/H, \emptyset/L_t, \emptyset/S, \emptyset/A]$$

The example of Figure 2 is a valid program.

2.5 Soundness

In this subsection we define a notion of *valid program state* and we prove that valid states are legal states. We then prove that program states reached by valid programs are valid, by proving that the initial state is valid and that execution steps preserve validity. It follows that valid programs are legal programs and therefore they are data-race-free.

A program state is *valid* if each thread state is *consistent* and each activation record is *valid*. The latter means that the continuation verification condition of the activation record's continuation holds with respect to the activation record's postcondition. An activation record's validity is independent of actions performed by other activation records. We prove this using the notion of *activation record access sets*. We prove that an activation record's validity depends only on the state of the objects in its access set, and actions of other activation records are non-interfering with respect to this access set.

We use the following shorthands. We denote the components of a thread-relevant state θ as A_θ , L_θ , S_θ , and A_θ . That is, we have

$$\theta = (A_\theta, L_\theta, S_\theta, A_\theta)$$

Furthermore, we use $A_{\text{req}}(\text{call}, \theta)$ to denote the required access set of call call in pre-state θ :

$$A_{\text{req}}(o.m(\bar{v}), \theta) = \mathcal{A}(\text{pre}(m)[o/\mathbf{this}, \bar{v}/\bar{x}, H_\theta/H, L_\theta/L_t, S_\theta/S])$$

We use $\text{post}(\text{call}, \theta)$ to denote the postcondition of call call with respect to pre-state θ :

$$\text{post}(o.m(\bar{v}), \theta) = \text{post}(m)[o/\mathbf{this}, \bar{v}/\bar{x}, H_\theta/H^{\text{old}}, L_\theta/L_t^{\text{old}}, S_\theta/S^{\text{old}}]$$

Definition 21. An activation record's access set is recursively defined as follows:

- The top (i.e., active) activation record's access set is the thread's access set minus the other activation records' access sets.
- The access set of an activation record that is suspended while waiting for a call to return, is the pre-state thread access set, minus the access sets of transitive caller activation records, minus the call's required access set.

Formally:

$$\begin{aligned}
t &= (\text{tid}, \mathbf{A}, (\bar{s}_1) \cdot (x_2 := \mathbf{receive} [\theta_2] \text{ call}_2; \bar{s}_2) \\
&\quad \dots \cdot (x_n := \mathbf{receive} [\theta_n] \text{ call}_n; \bar{s}_n) \cdot \epsilon) \\
&\quad \quad \quad \downarrow \\
\forall i \in \{1, \dots, n\} \bullet \mathbf{A}_{\text{ar}(i)}(t) &= \begin{cases} \mathbf{A} - \bigcup_{1 < j \leq n} \mathbf{A}_{\text{ar}(j)}(t) & \text{if } i = 1 \\ \mathbf{A}_{\theta_i} - \bigcup_{i < j \leq n} \mathbf{A}_{\text{ar}(j)}(t) - \mathbf{A}_{\text{req}}(\text{call}_i, \theta_i) & \text{if } i > 1 \end{cases}
\end{aligned}$$

Definition 22. A thread state is consistent with respect to a given lockset if all of the following thread state consistency requirements hold:

- TSC1: The thread's lockset is equal to the set of objects that appear in **unlock** statements in the thread's activation records' continuations
- TSC2: No object appears more than once in an **unlock** statement
- TSC3: A caller's pre-state lockset is equal to the set of the target objects of the **unlock** statements in the call's continuation plus transitive caller continuations
- TSC4: Each non-top activation record's required access set is included in its pre-state activation record access set
- TSC5: Each non-top activation record's pre-state access set includes the access sets of transitive caller activation records
- TSC6: The thread's access set includes the non-top activation records' access sets
- TSC7: Each object bound in a given activation record is in the access set of one of the activation record's reflexive-transitive callees
- TSC8: An object bound by an **unlock** block is not bound by any enclosing block in the same activation record or in any transitive caller
- TSC9: A catch block's pre-state bound set is equal to the set of the objects bound by transitively enclosing blocks (including those in transitive callers)
- TSC10: A catch block's pre-state lockset is equal to the set of the target objects of the transitively enclosing unlock blocks (including those in transitive callers)

Formally:

$$\begin{aligned}
& L_t \vdash \text{consistent}(t) \\
& \quad \Downarrow \\
& \quad t \text{ is of the form} \\
& (\text{tid}, \mathbf{A}, (\bar{s}_1) \cdot (x_2 := \mathbf{receive} [\theta_2] \text{ call}_2; \bar{s}_2) \\
& \quad \dots \cdot (x_n := \mathbf{receive} [\theta_n] \text{ call}_n; \bar{s}_n) \cdot \epsilon) \\
& \quad \text{where} \\
& L_t = \{o \mid (\exists i \in \{1, \dots, n\} \bullet \bar{s}_i = \dots \mathbf{unlock} o; \dots)\} \\
& \quad \wedge \\
& \quad \neg(\bar{s}_1 \dots \bar{s}_n = \dots \mathbf{unlock} o; \dots \mathbf{unlock} o; \dots) \\
& \quad \wedge \\
& (\forall i \in \{2, \dots, n\} \bullet L_{\theta_i} = \{o \mid (\exists j \in \{i, \dots, n\} \bullet \bar{s}_j = \dots \mathbf{unlock} o; \dots)\}) \\
& \quad \wedge \\
& \quad (n = 1 \\
& \quad \vee \\
& \quad (\mathbf{A}_{\text{req}}(\text{call}_n, \theta_n) \subseteq \mathbf{A}_{\theta_n} \\
& \quad \wedge \\
& \quad (\forall i \in \{2, \dots, n-1\} \bullet \\
& \quad \mathbf{A}_{\theta_{i+1}} - \mathbf{A}_{\text{req}}(\text{call}_{i+1}, \theta_{i+1}) \subseteq \mathbf{A}_{\theta_i} \\
& \quad \wedge \\
& \quad \mathbf{A}_{\text{req}}(\text{call}_i, \theta_i) \subseteq \mathbf{A}_{\theta_i} - (\mathbf{A}_{\theta_{i+1}} - \mathbf{A}_{\text{req}}(\text{call}_{i+1}, \theta_{i+1}))) \\
& \quad \wedge \\
& \quad \mathbf{A}_{\theta_2} - \mathbf{A}_{\text{req}}(\text{call}_2, \theta_2) \subseteq \mathbf{A}) \\
& \quad \wedge \\
& \quad \vdots)
\end{aligned}$$

Notice that if a thread state is consistent, then its activation records' access sets partition the thread's access set.

Definition 23. An activation record's postcondition $Q_{\text{ar}(i)}(t)$ is the postcondition of the call stored in the **receive** statement in the caller's continuation, or **true** if the activation record has no caller.

Formally:

$$\begin{aligned}
& t = (\text{tid}, \mathbf{A}, (\bar{s}_1) \cdot (x_2 := \mathbf{receive} [\theta_2] \text{ call}_2; \bar{s}_2) \\
& \quad \dots \cdot (x_n := \mathbf{receive} [\theta_n] \text{ call}_n; \bar{s}_n) \cdot \epsilon) \\
& \quad \Downarrow \\
& \forall i \in \{1, \dots, n\} \bullet Q_{\text{ar}(i)}(t) = \begin{cases} \text{post}(\text{call}_{i+1}, \theta_{i+1}) & \text{if } i < n \\ \mathbf{true} & \text{if } i = n \end{cases}
\end{aligned}$$

Note: the caller may be in another thread in the case of a parallel execution statement.

Definition 24. The failure verification condition of an enclosing block list with respect to a given postcondition is defined as follows:

$$\begin{aligned}
& \text{vppc}(b \cdot \bar{b}, Q) = \text{vpc}(b \cdot \bar{b}, Q) \wedge \text{vppc}(\bar{b}, Q) \\
& \text{vppc}(\epsilon, Q) = \mathbf{true}
\end{aligned}$$

Definition 25. The block invariant of an enclosing block list is the conjunction of the block invariant of the catch blocks.

$$\begin{aligned} \text{blockinvariant}(\epsilon) &= \text{true} \\ \text{blockinvariant}((B, \text{unlock } o; \bar{s}) \cdot \bar{b}) &= \text{blockinvariant}(\bar{b}) \\ \text{blockinvariant}((B, \text{catch } [\theta, \{\bar{x}\}, \phi] \{ \bar{s}_1 \} \bar{s}_2) \cdot \bar{b}) &= \phi \wedge \text{blockinvariant}(\bar{b}) \end{aligned}$$

Definition 26. An activation record is valid if the verification condition of its normal continuation with respect to its postcondition holds under the current heap, lock set, and shared set, and under the activation record's access set and the thread's bound set, and each enclosing catch block's block invariant holds, and the failure verification condition of its enclosing block list holds. Formally:

$$\begin{aligned} t &= (\text{tid}, A, (V_1, \bar{s}_1, \bar{b}_1) \cdot \dots \cdot (V_n, \bar{s}_n, \bar{b}_n) \cdot \epsilon) \\ &\Downarrow \\ &(\forall i \in \{1, \dots, n\} \bullet \\ &\quad \text{H}, \text{L}, \text{S} \vdash \text{valid}_{\text{ar}(i)}(t)) \\ &\Updownarrow \\ &(\mathfrak{J}, \text{H}, \text{L}^{-1}(\text{tid}), \text{S}, \text{A}_{\text{ar}(i)}(t), \text{B}(t), V_i \models \text{vc}(\bar{s}_i, \bar{b}_i, Q_{\text{ar}(i)}(t))) \\ &\quad \wedge \\ &\quad \mathfrak{J}, \text{H}, \text{L}^{-1}(\text{tid}), \text{S}, \text{A}_{\text{ar}(i)}(t), \text{B}(t), V_i \models \text{blockinvariant}(\bar{b}_i) \\ &\quad \wedge \\ &\quad \mathfrak{J}, V_i \models \text{vppc}(\bar{b}_i, Q_{\text{ar}(i)}(t))) \end{aligned}$$

Definition 27. A program state is valid if it is well-formed, each thread state is well-formed and consistent, and each activation record is valid.

$$\begin{aligned} &\text{valid}(\text{H}, \text{L}, \text{S}, \text{T}) \\ &\quad \Updownarrow \\ &\quad \text{wf}(\text{H}, \text{L}, \text{S}, \text{T}) \\ &\quad \quad \wedge \\ &\quad \quad (\forall t \in \text{T} \bullet t = (\text{tid}, A, r_1 \cdot \dots \cdot r_n)) \\ &\quad \quad \Downarrow \\ &\quad \quad \vdash (\text{H}, \text{L}^{-1}(\text{tid}), \text{S}, A) : \text{ok} \wedge \text{L}^{-1}(\text{tid}) \vdash \text{consistent}(t) \\ &\quad \quad \quad \wedge \\ &\quad \quad \quad (\forall i \in \{1, \dots, n\} \bullet \text{H}, \text{L}, \text{S} \vdash \text{valid}_{\text{ar}(i)}(t))) \end{aligned}$$

Theorem 6. A valid program state is a legal program state.

Proof. One can easily prove that each thread is in a legal state by performing a case analysis on the first statement of the normal continuation of the top activation record and for each case using the validity of the top activation record.

In case the first statement of the normal continuation is a throw statement, legality is trivial except if the top activation record's enclosing block list starts with an **unlock** o ; continuation; in this case we need to prove that o is accessible and locked by the current thread. It is accessible because of the thread state consistency requirement saying that if an object is bound by an activation record, it is in the access set of one of the activation record's reflexive-transitive callers. It is locked because of the thread state consistency requirement saying that each object mentioned in an **unlock** continuation is in the thread's lockset. \square

Theorem 7. *In a valid program π , the small step relation \rightarrow preserves validity.*

$$\forall \sigma_1, \sigma_2 \bullet (\text{valid}(\sigma_1) \wedge \sigma_1 \rightarrow \sigma_2) \Rightarrow \text{valid}(\sigma_2)$$

Proof. By case analysis on the step. Note that preservation of well-formedness is given by Theorem 1. We refer to the thread that performs the step as the current thread and the top activation record of the current thread as the current activation record. For each step rule, we have to prove that in σ_2 , thread states are well-formed and consistent and activation records are valid.

Each step changes the current activation record's continuation. We only note other changes. Also, we only detail the argument for preservation of validity of the current activation record if it does not follow easily from the verification condition.

- **Case IF, ASSERT.** The step changes only the continuation of the current activation record. Therefore, thread state consistency and validity of the other activation records is preserved trivially.
- **Case WRITE.** The step changes the heap at some location $o.f$, and the current activation record's continuation. Thread state consistency depends on neither so it is preserved trivially. By VC-WRITE, we have that o is in the current activation record's access set. Well-formedness of the thread-relevant state is preserved since the value written into the field is either **null** or a shared object. Since activation record access sets are disjoint and, as a result of the locality of continuation verification conditions (Theorem 5), activation record validity depends only on heap locations in the activation record's access set, validity of other activation records is preserved trivially.
- **Case SHARE.** The step changes the shared set and the current thread's access set. Since the object being shared was in the current activation record's access set, it is not in the access set of any other activation record of the current thread so the access set still contains those. This establishes thread state consistency. The validity of an activation record is preserved by sharing an object that is not in its access set, and by removing from the thread's access set an object that is not in the activation record's access set, so the validity of non-current activation records is preserved.
- **Case CALL.** The step changes the current activation record's continuation and adds a new activation record. Since the precondition holds, its required access set is contained in the caller's pre-state access set. Therefore, thread state consistency is preserved. Validity of existing non-current activation records is preserved trivially. Since the program is valid, the method being called is valid, and the method's body is valid in any state that satisfies the precondition. Note that the new activation record's access set is equal to the precondition's required access set.
- **Case RETURN.** The step replaces the caller and callee activation records with an activation record containing the continuation of the call. Validity of the caller activation record implies that the call's continuation is valid in any thread state that a) differs from the current state only as allowed by the method's frame condition and b) satisfies the postcondition. Validity

of the callee implies that the postcondition holds in state σ_1 . Therefore the call's continuation is valid in state σ_1 and, therefore, in state σ_2 .

- **Case NEWTHREAD.** The step adds a new thread with a single activation record, and removes the target object from the creating thread's access set. Thread state consistency of the new thread is trivial. The new activation record is valid since the method it executes is valid and its precondition, which by well-formedness of the program must be $L_t = \emptyset \wedge \mathbf{this} \in A$, is satisfied.
- **Case READ.** The step changes only the current activation record's continuation. Validity follows trivially from VC-READ.
- **Case NEW.** The step adds an object to the heap domain and the thread's access set. Since by well-formedness access sets contain only allocated objects, access sets remain disjoint.
- **Case SYNCHRONIZED.** The step adds an object to the lock set and the access set. Since by well-formedness the free set is disjoint from access sets and the object was in the free set, access sets remain disjoint.
- **Case UNLOCK.** The step removes an object from the thread's access set. Since it was in the current activation record's access set, no other activation records are affected. Thread state consistency requirement TSC7 is preserved thanks to requirement TSC8.
- **Case CATCH.** Trivial.
- **Case FAIL.** The step replaces the top activation record's continuation with a throw statement. Therefore, the continuation is trivially valid.
- **Case FAIL-UNLOCK.** Is valid for the same reason as Case UNLOCK.
- **Case FAIL-CATCH.** Trivial.
- **Case FAIL-CALL.** The step removes the top activation record and replaces the caller activation record's continuation with a throw statement. Trivially valid.
- **Case PAR.** The step pushes a new activation record onto the current thread, and creates a new thread.
- **Case PAR-COMPLETE.** The step is enabled when both branches of the parallel composition are complete. The right-hand thread is deleted, as is the top activation record of the left-hand thread.
- **Case PACK.** The step sets the argument's inv bit.
- **Case UNPACK.** The step clears the argument's inv bit.

This concludes the proof. □

Theorem 8. *A valid program is a legal program.*

Proof. The initial state of a valid program is a valid state. It follows, by Theorem 7, that all reachable states are valid. Therefore, by Theorem 6, all reachable states are legal. Therefore, the program is legal. \square

Theorem 9 (Main Theorem). *Valid programs are data-race-free.*

Proof. By combining Theorem 8 and Theorem 3. \square