Electronic Notes in Theoretical Computer Science 89 No. 2 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 20 pages

Formalizing Java-MaC

Usa Sammapun [1] Raman Sharykin [2] Margaret DeLap [1]
Myong Kim [1] Steve Zdancewic [1]

University of Pennsylvania

Abstract

The Java-MaC framework is a run-time verification system for Java programs that can be used to dynamically test and enforce safety policies. This paper presents a formal model of the Java-MaC safety properties in terms of an operational semantics for Middleweight Java, a realistic subset of full Java. This model is intended to be used as a framework for studying the correctness of Java-MaC program instrumentation, optimizations, and future experimentation with run-time monitor expressiveness. As a preliminary demonstration of this model's applicability for these tasks, the paper sketches a correctness result for a simple program instrumentation scheme.

1 Introduction

A *run-time monitor* is a process that observes the execution of a target system, verifies that the system meets a given safety property, and takes action when the system violates the property. One might use run-time monitors to halt a program that tries to gain unauthorized access to a file or to notify an administrator when a target variable exceeds some threshold. Because of their generality and potential for efficient implementation, run-time monitors are an attractive way to enforce security and other safety-critical properties.

Java-MaC [12] is an instance of a *Monitoring and Checking* (MaC) framework developed at the University of Pennsylvania [13,11]. The goal of the project is to apply run-time verification technology to Java programs.

Given a target Java program, a Java-MaC user can specify a safety property in terms of *events* and *conditions*. Events are instantaneous incidents such as variable updates or the start or end of a method call. Conditions are propositions about the program that may be true or false for a duration of time, such as `Train.speed > 20`. The language for describing Java-MaC

[1] Email: {usa, delap, myongkim, stevez}@seas.upenn.edu
[2] Email: rsharyki@sas.upenn.edu

events and conditions is derived from the source Java program, and more complex safety properties can be built from those primitive events and conditions. For example, the user might want to be informed when the `Train.brake()` method is invoked whenever the speed is greater than 20. The Java-MaC specification of this event is:

```
StartM(Train.brake) when (Train.speed > 20)
```

Given a safety-property specification, the Java-MaC system automatically instruments the bytecode of the target Java program. The instrumented bytecode, called a *filter*, generates event messages that are sent to an event recognizer and a runtime checker, which run in parallel with the target system. The bytecode instrumentation is done statically. At runtime, the filter sends the information being probed to the event recognizer and runtime checker that monitor and react to the sequence of generated events.

To date, the event recognizer and the runtime checker have been well studied by the MaC researchers. However, the semantics of Java-MaC specifications and the instrumentation process have not. This paper presents a formal semantics for the Java-MaC framework. It gives a precise operational semantics for a core, but realistic fragment of the Java language and shows how that operational model can be used to give meaning to the Java-MaC specification language. This strategy offers a number of benefits:

- Existing work on defining the Java-MaC semantics is presented at the Java bytecode level of abstraction. However, because Java-MaC specifications are given in terms of source-program expressions, there is a semantic gap between what the Java-MaC user is reasoning about (Java programs and high-level specifications) and what the underlying implementation does. Giving a precise definition of Java-MaC specifications makes the meaning of Java-MaC specifications with respect to the source program more apparent.
- In a related vein, previous work on the Java-MaC framework has described the program instrumentation process at the level of Java bytecode [11]. That work also gives an informal argument that the instrumented code behaves the same as the original program (modulo certain timing and synchronization considerations). By giving a source-level semantics to Java-MaC, this work opens up the possibility of formally verifying that the bytecode instrumentation process corresponds with the source-level semantics.
- Working with a precise operational semantics for Java has revealed ambiguities in the definition of Java-MaC semantics. These ambiguities arise because of Java-specific treatment of things like variable scoping and inheritance. Using a formal operational model highlights the places where different design choices can be made.
- This operational approach lays the groundwork for future experimentation with Java-MaC designs. For example, one could pursue extensions of the Java-MaC framework that permit monitoring of objects or relax Java-MaC's current restrictions on reference aliasing.

To demonstrate the benefits of this approach, this paper gives a toy Java-level instrumentation algorithm that generates (some of) the events needed by the Java-MaC event recognizer and runtime checker. It also sketches how to establish a simulation relation between the original program and the instrumented program to prove that the instrumentation process is correct.

The next section introduces Middleweight Java, the subset of Java on which our semantics is based. Section 3 gives the semantics of Java-MaC events and conditions. Section 4 gives the instrumentation algorithm and states the appropriate simulation result. Section 5 discusses related work, and 6 concludes with some of the lessons learned from formalizing Java-MaC and some directions for future work.

2 Middleweight Java

This section describes the syntax and operational semantics of Middleweight Java (MJ), a fragment of Java amenable to formal proof and analysis techniques. This variant of MJ closely follows prior work on Middleweight Java [5] and Featherweight Java [10], but it extends those languages with static fields and methods, which are needed for Java-MaC safety property specifications.

Figure 1 gives the grammar for MJ programs. The syntax is a strict subset of the full Java language [9]; any valid MJ program is also a valid Java program. The intention is that the evaluation of an MJ program captures the behavior of the corresponding Java program.

The syntax is defined in terms of five kinds of identifiers, which are just strings. Each kind of identifier is drawn from a set given below:

$\mathcal{C}$ – Class names: C, D, `Object`, etc.
$\mathcal{M}$ – Method names: m, `main`, etc.
$\mathcal{F}$ – Field names: f, `field`, etc.
$\mathcal{V}$ – Variable names: x, y, `this`, etc.
$\mathcal{R}$ – References (object identifiers): r, r_1, r_2, etc.

MJ expressions can have primitive type `boolean` or an object type C, and the syntactic class T ranges over expression types. For simplicity, the formalism here does not consider other primitive Java types such as `int` or `byte`, but they could be added straightforwardly.

Class declarations, as in Java, contain a suite of field declarations $\bar{fd}$ and method declarations $\bar{md}$. Here, and throughout the paper, an overbar is used to denote a possibly empty vector of program phrases. For example, $\bar{x} = x_1 x_2 \ldots x_n$ for some $n \geq 0$.

Field declarations consist of a type T and the name of the field f, optionally including the keyword `static` to indicate that the field is shared among all instances of the class. Method declarations include a return type τ that is either an expression type T or the Java keyword `void`. There may be zero or more formal parameters in a method declaration, and the body of the method

T	::=	`boolean` $\mid$ C	Expression type
τ	::=	`void` $\mid$ T	Return type
cd	::=	`class` C `extends` C `{`$\bar{fd}$ $\bar{md}$`}`	Class decl.
fd	::=	$T f$`;` $\mid$ `static` $T f$`;`	Field decl.
md	::=	τ m`(`T_1 x_1`,` $\ldots$ `,`T_n x_n`){`$\bar{s}$`}`	Method decl.
	$\mid$	`static` τ m`(`T_1 x_1`,` $\ldots$ `,`T_k x_n`){`$\bar{s}$`}`	
e	::=	x	Variables
	$\mid$	`null` $\mid$ `true` $\mid$ `false`	Basic values
	$\mid$	$e.f$ $\mid$ $C.f$	Field access
	$\mid$	e `==` e $\mid$ e `||` e $\mid$ $\ldots$	Primitive operations
	$\mid$	e `instanceof` C	Instanceof
	$\mid$	`(`C`)` e	Cast
	$\mid$	pe	Promotable expression
pe	::=	`new` C`()`	Object creation
	$\mid$	$e.m$`(`$\bar{e}$`)` $\mid$ $C.m$`(`$\bar{e}$`)`	Method invocation
s	::=	`;`	No-op
	$\mid$	pe`;`	Promoted expression
	$\mid$	T x `=` e`;`	Local variable decl.
	$\mid$	x `=` e`;`	Local variable update
	$\mid$	$e.f$ `=` e`;` $\mid$ $C.f$ `=` e`;`	Field update
	$\mid$	`if (`e`) {`$\bar{s}$`} else {`$\bar{s}$`}`	Conditional
	$\mid$	`return` e`;`	Return
	$\mid$	`{`$s_1 \ldots s_n$`}`	Block

Fig. 1. Syntax

consists of a vector of statements $\bar{s}$. As with fields, methods may be static, but for simplicity, MJ does not include other field and method access modifiers such as `public` or `final`.

The simplest expressions in MJ consist of variables, x, primitive values `true` and `false`, and `null`. More complex expressions can be built via dynamic field access $e.f$ and static field access $C.f$. Note that unlike Java, MJ requires that all dynamic field accesses be performed with respect to an object reference, so MJ programs use `this.`f to access the local field f. A similar restriction holds for method invocation.

Other MJ expressions include the `instanceof` operator to test class membership of an object, and a dynamic cast mechanism. Standard `boolean` operations such as equality and logical or are also included in MJ.

MJ and Java permit promotable expressions to be treated as either value-returning expressions or as program statements. Object creation (via `new`) and dynamic and static method invocation are promotable expressions. Again for simplicity, the version of MJ used in this paper does not permit object constructors; instead fields of type `boolean` are initialized to `false` and fields of object type are initialized to `null`. Other variants of MJ [5] provide a full treatment of object constructors.

Statements in MJ include the no-op '`;`' statement, as well as the standard

$$\Delta_m(C,m) \stackrel{\text{def}}{=} \begin{cases} \bar{T} \to \tau & \text{if } md_i = [\texttt{static}]\ \tau\ m\texttt{(}\bar{T}\ \bar{x}\texttt{)\{...\}},\ i \in \{1 \ldots n\} \\ \Delta_m(C',m) & \text{if } m \notin \{md_1 \ldots md_n\} \end{cases}$$

$$\Delta_f(C,f) \stackrel{\text{def}}{=} \begin{cases} T & \text{if } fd_i = [\texttt{static}]\ T\ f,\ i \in \{1 \ldots k\} \\ \Delta_f(C',f) & \text{if } f \notin \{fd_1 \ldots fd_k\} \end{cases}$$

$$\text{mbody}(C,m) \stackrel{\text{def}}{=} \begin{cases} (\bar{x},\bar{s}) & \text{if } md_i = \tau\ m\texttt{(}\bar{T}\ \bar{x}\texttt{)\{}\bar{s}\texttt{\}},\ i \in \{1 \ldots n\} \\ \text{mbody}(C',m) & \text{if } m \notin \{md_1 \ldots md_n\} \end{cases}$$

where $\texttt{class}\ \ C\ \texttt{extends}\ \ C'\ \texttt{\{}fd_1 \ldots fd_k\ md_1 \ldots md_n\texttt{\}} \in prog$

Fig. 2. Class tables and method body lookup

local variable declaration, variable and field assignment, conditional, and return statement. New blocks can be introduced as in Java by using `{...}`.

Although MJ is a faithful subset of Java, it is missing some features that we leave to future work. Some of them, like arrays, should be straightforward to include. Others, like exceptions and multithreading, will require more substantial changes to the operational semantics presented below.

2.1 MJ static semantics

An MJ *program* is a sequence of class declarations followed by a statement: $prog = cd_1 cd_2 \ldots cd_n\,;s$. A given program *prog* induces a subclass relation between class names, written $C \prec D$. The subclass relation is defined to be the reflexive, transitive closure of the `extends` relation found in class declarations. Class C `extends` class D if $\texttt{class}\ \ C\ \texttt{extends}\ \ D\ \texttt{\{...\}}$ appears in *prog*. For a valid program, the subclass relation forms a tree rooted at the class named `Object`.

A well-formed program *prog* has an associated class table Δ comprising two partial functions: Δ_f maps class names and field names to their types and Δ_m maps class names and method names to their types. For example, consider the following class declaration:

```
class C extends Object {
  boolean b;
  void foo(boolean x) {...} }
```

The class table for this program has $\Delta_f(\texttt{C},\texttt{b}) = \texttt{boolean}$ and $\Delta_m(\texttt{C},\texttt{foo}) = \texttt{boolean} \to \texttt{void}$. Method types, as for `foo`, are of the form $\bar{T} \to \tau$.

Given a well formed program *prog*, the partial function $\text{mbody}(C,m)$ extracts the formal parameters and method body of the method named m from the class C if such a method exists. Inheritance means that both the class tables and method body functions are defined inductively on the class hierarchy, as shown in Figure 2.

MJ has a static type system derived from Java's. The typing judgments for expressions are of the form $\Delta;\Gamma \vdash e : T$. Γ is a finite partial map from variable

names to expression types. This judgment says that expression e has type T in the context of the class table Δ and typing environment Γ. Similarly, rules for typechecking statements have the form $\Delta; \Gamma \vdash s : \tau$. Space constraints prevent the full type system from being reproduced here, see Bierman et al. for full details [5].

2.2 Dynamic semantics

This section defines an operational semantics for MJ programs in terms of a transition relation between abstract machine states. Because Java has relatively complex rules for allocating stack space for local variables and for allocating static and dynamic objects in the heap, the states are themselves relatively complex. A *state* or *configuration* σ is a five-tuple $(\mathbb{S}, \mathbb{H}, V, G, K)$. Figure 3 gives the formal syntax for MJ states.

$\mathbb{S}$ is the static portion of the heap—it contains a mapping from class names and their static fields to the values stored in those fields. Formally, $\mathbb{S}$ is a finite partial function from $\mathcal{C} \times \mathcal{F}$ to values. Values are `null`, `true`, `false`, and object references $r \in \mathcal{R}$. For notational consistency, all finite partial functions like $\mathbb{S}$ in the operational semantics use the blackboard-bold font. The symbol $\rightharpoonup$ indicates a finite partial function space, and $\emptyset$ denotes the everywhere undefined partial function. If $\mathbb{P}$ is a partial function, then $\mathbb{P}[x \mapsto y]$ is a new partial function that agrees with $\mathbb{P}$ except at x, which is mapped to y.

$\mathbb{H}$ is the dynamic heap. It contains a mapping from object references $\mathcal{R}$ to heap objects. A heap object *ho*, which represents an instance of a class, is a pair of the class name C and a field function $\mathbb{F}$. The field function for an instance of an object maps the names of the object's nonstatic fields to the values stored in the fields.

V is the stack of local variables. When a method m in class C is invoked, a new method scope is pushed on to V to produce a new local variable stack $(M)^C_m :: V$. Because a single method m can contain several different lexical blocks each containing its own local variables, M, a method scope, is itself a stack of block scopes. Each block scope $\mathbb{B}$ maps local variable names to the values stored in them. To evaluate or update a local variable x in method scope M it is necessary to traverse the stack of blocks in M, as shown in the following definitions:

$$\text{eval}((\mathbb{B} :: M), x) \stackrel{\text{def}}{=} \begin{cases} \mathbb{B}(x) & \text{if } x \in \text{dom}(\mathbb{B}) \\ \text{eval}(M, x) & \text{otherwise} \end{cases}$$

$$\text{update}((\mathbb{B} :: M), x \mapsto v) \stackrel{\text{def}}{=} \begin{cases} \mathbb{B}[x \mapsto v] :: M & \text{if } \mathbb{B}(x) = v' \\ \mathbb{B} :: \text{update}(M, x \mapsto v) & \text{otherwise} \end{cases}$$

G is the state's frame stack, which consists of a stack of closed or open frames. Closed frames K are statements or expressions whose evaluation can proceed without first computing the value of subexpressions embedded within them. Open frames, by contrast, are expressions and statements with a "hole" $[\cdot]$ that must be filled with a value before they may be evaluated. As an exam-

$$
\begin{array}{rcll}
\sigma & ::= & (\mathbb{S}, \mathbb{H}, V, G, K) & \text{Machine state} \\
v & ::= & \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid r & \text{Value} \\
V & ::= & (M)_m^C :: V \mid \cdot & \text{Variable stack} \\
M & ::= & \mathbb{B} :: M \mid \cdot & \text{Method scope} \\
ho & ::= & (C, \mathbb{F}) & \text{Heap object} \\
\mathbb{S} & : & \mathcal{C} \times \mathcal{F} \rightharpoonup \text{Value} & \text{Static heap} \\
\mathbb{H} & : & \mathcal{R} \rightharpoonup \text{Heap object} & \text{Heap} \\
\mathbb{B} & : & \mathcal{V} \rightharpoonup \text{Value} & \text{Block scope} \\
\mathbb{F} & : & \mathcal{F} \rightharpoonup \text{Value} & \text{Field map} \\
G & ::= & F :: G \mid \cdot & \text{Frame stack} \\
F & ::= & K \mid H & \text{Frame} \\
K & ::= & \bar{s} \mid \texttt{return}\ e\texttt{;} \mid \texttt{\{\}} \mid e & \text{Closed frame} \\
H & ::= & [\cdot] \mid H.f \mid H.m(\bar{e}) \mid v.m(\bar{v}, H, \bar{e}) & \text{Open frame} \\
 & \mid & H \texttt{ || } e \mid v \texttt{ || } H & \\
 & \mid & H \texttt{ == } e \mid v \texttt{ == } H \mid \texttt{(}C\texttt{)}\ H \mid H\ \texttt{instanceof}\ C & \\
 & \mid & H\texttt{;} \mid T\ x \texttt{ = } H\texttt{;} \mid x \texttt{ = } H\texttt{;} \mid H.f \texttt{ = } e\texttt{;} \mid v.f \texttt{ = } H\texttt{;} & \\
 & \mid & \texttt{if (}H\texttt{) \{}\bar{s}_1\texttt{\} else \{}\bar{s}_2\texttt{\}} \mid \texttt{return}\ H\texttt{;} \mid H\texttt{;} &
\end{array}
$$

Fig. 3. Dynamic state

ple, consider the evaluation of the expression $e =$ `(true || false) || false`. This expression can be decomposed into an open frame $H = [\cdot]$ `||` `false` and the closed frame $K =$ `true || false` such that $e = H[K]$. (The notation $H[x]$ means the closed frame H where the hole has been replaced by x.) In the operational semantics this closed frame K evaluates to `true`, and so e reduces to $H[$`true`$] =$ `true || false`, which can then be further reduced in subsequent steps. Intuitively, the stack of frames G keeps track or the work yet to be done by the computation.

The final component of the state is a closed frame K, which is the statement or expression currently being evaluated.

Figures 4 and 5 give the complete operational semantics for MJ as a transition relation between states. Most of the rules follow directly from Java semantics. For example, rule IfTrue shows that when the guard of an `if` statement evaluates to `true` the evaluation continues inside the first branch.

Variable reading and writing use the eval and update functions to access the local variable x in the current method scope, as shown in VarRead and VarWrite. Variable declaration (VarDecl) adds a new mapping to the topmost block in scope. BlkBegin creates a new block scope, which is initially empty, when the block is entered. That rule also pushes the empty block `{}` onto the frame stack. As shown in BlkEnd, when the empty block is reached the block scope is popped off the stack.

FldRead shows that to read a field f of the object through reference r first the heap object is obtained from $\mathbb{H}$ and then the instance field function is used to lookup the value. To read a static field, the static heap $\mathbb{S}$ is used instead (SFldRead). FldWrite and SFldWrite update the appropriate parts of the heap to point to the new value.

VarRead $(\mathbb{S}, \mathbb{H}, (M)^C_m :: V, G, x) \to (\mathbb{S}, \mathbb{H}, (M)^C_m :: V, G, v)$
when $\text{eval}(M, x) = v$

VarWrite $(\mathbb{S}, \mathbb{H}, (M)^C_m :: V, G, x \texttt{ = } v\texttt{;}) \to (\mathbb{S}, \mathbb{H}, (\text{update}(M, x \mapsto v))^C_m :: V, G, \texttt{;})$
when $\text{eval}(M, x) = v'$

VarDecl $(\mathbb{S}, \mathbb{H}, (\mathbb{B} :: M)^C_m :: V, G, T\ x \texttt{ = } v\texttt{;}) \to (\mathbb{S}, \mathbb{H}, (\mathbb{B}' :: M)^C_m :: V, G, \texttt{;})$
when $x \notin \text{dom}(\mathbb{B} :: M)$ and $\mathbb{B}' = \mathbb{B}[x \mapsto v]$

BlkBegin $(\mathbb{S}, \mathbb{H}, (M)^C_m :: V, G, \texttt{\{}\bar{s}\texttt{\}}) \to (\mathbb{S}, \mathbb{H}, (\emptyset :: M)^C_m :: V, \texttt{\{\}} :: G, \bar{s})$

BlkEnd $(\mathbb{S}, \mathbb{H}, (\mathbb{B} :: M)^C_m :: V, G, \texttt{\{\}}) \to (\mathbb{S}, \mathbb{H}, (M)^C_m :: V, G, \texttt{;})$

IfTrue $(\mathbb{S}, \mathbb{H}, V, G, \texttt{if (true) \{}\bar{s}_1\texttt{\} else \{}\bar{s}_2\texttt{\}}) \to (\mathbb{S}, \mathbb{H}, V, G, \texttt{\{}\bar{s}_1\texttt{\}})$

IfFalse $(\mathbb{S}, \mathbb{H}, V, G, \texttt{if (false) \{}\bar{s}_1\texttt{\} else \{}\bar{s}_2\texttt{\}}) \to (\mathbb{S}, \mathbb{H}, V, G, \texttt{\{}\bar{s}_2\texttt{\}})$

IofNull $(\mathbb{S}, \mathbb{H}, V, G, \texttt{null instanceof } C) \to (\mathbb{S}, \mathbb{H}, V, G, \texttt{true})$

IofTrue $(\mathbb{S}, \mathbb{H}, V, G, r \texttt{ instanceof } C) \to (\mathbb{S}, \mathbb{H}, V, G, \texttt{true})$
when $\mathbb{H}(r) = (D, \mathbb{F})$ and $D \prec C$

IofFalse $(\mathbb{S}, \mathbb{H}, V, G, r \texttt{ instanceof } C) \to (\mathbb{S}, \mathbb{H}, V, G, \texttt{false})$
when $\mathbb{H}(r) = (D, \mathbb{F})$ and $D \not\prec C$

FldRead $(\mathbb{S}, \mathbb{H}, V, G, r.f) \to (\mathbb{S}, \mathbb{H}, V, G, v)$
when $\mathbb{H}(r) = (T, \mathbb{F})$ and $\mathbb{F}(f) = v$

SFldRead $(\mathbb{S}, \mathbb{H}, V, G, C.f) \to (\mathbb{S}, \mathbb{H}, V, G, v)$
when $\mathbb{S}(C, f) = v$

FldWrite $(\mathbb{S}, \mathbb{H}, V, G, r.f \texttt{ = } v\texttt{;}) \to (\mathbb{S}, \mathbb{H}', V, G, \texttt{;})$
when $\mathbb{H}(r) = (T, \mathbb{F})$ and $\mathbb{F}' = \mathbb{F}[f \mapsto v]$ and $\mathbb{H}' = \mathbb{H}[r \mapsto (T, \mathbb{F}')]$

SFldWrite $(\mathbb{S}, \mathbb{H}, V, G, C.f \texttt{ = } v\texttt{;}) \to (\mathbb{S}', \mathbb{H}, V, G, \texttt{;})$
when $\mathbb{S}(C, f) = v'$ and $\mathbb{S}' = \mathbb{S}[(C, f) \mapsto v]$

Cast $(\mathbb{S}, \mathbb{H}, V, G, \texttt{(}C\texttt{)}\ r) \to (\mathbb{S}, \mathbb{H}, V, G, r)$
when $\mathbb{H}(r) = (D, \mathbb{F})$ and $D \prec C$

CastNull $(\mathbb{S}, \mathbb{H}, V, G, \texttt{(}C\texttt{) null}) \to (\mathbb{S}, \mathbb{H}, V, G, \texttt{null})$

Fig. 4. Operational semantics

New $(\mathbb{S}, \mathbb{H}, V, G, \texttt{new } C\texttt{()}) \to (\mathbb{S}, \mathbb{H}', V, G, r)$
when $r \notin \text{dom}(\mathbb{H})$, $\mathbb{F} = \text{initFields}(C)$, and $\mathbb{H}' = \mathbb{H}[r \mapsto (C, \mathbb{F})]$

EqTrue $(\mathbb{S}, \mathbb{H}, V, G, v \texttt{ == } v) \to (\mathbb{S}, \mathbb{H}, V, G, \texttt{true})$

EqFalse $(\mathbb{S}, \mathbb{H}, V, G, v \texttt{ == } v') \to (\mathbb{S}, \mathbb{H}, V, G, \texttt{false})$
when $v \neq v'$

Or $(\mathbb{S}, \mathbb{H}, V, G, v \texttt{ || } v') \to (\mathbb{S}, \mathbb{H}, V, G, v'')$
when $v, v' \in \{\texttt{true}, \texttt{false}\}$ and $v'' = v \vee v'$

Call $(\mathbb{S}, \mathbb{H}, V, G, r.m(\bar{v})) \to (\mathbb{S}, \mathbb{H}, (\mathbb{B})^C_m :: V, G, \bar{s})$
when $\mathbb{H}(r) = (C, \mathbb{F})$, $\text{mbody}(C, m) = (\bar{x}, \bar{s})$,
$\mathbb{B} = \{\texttt{this} \mapsto r, \bar{x} \mapsto \bar{v}\}$, and $\Delta(C, m) = \bar{T} \to T$

CallVoid $(\mathbb{S}, \mathbb{H}, V, G, r.m(\bar{v})) \to$
$(\mathbb{S}, \mathbb{H}, (\mathbb{B})^C_m :: V, (\texttt{return null;}) :: G, \bar{s})$
when $\mathbb{H}(r) = (C, \mathbb{F})$, $\text{mbody}(C, m) = (\bar{x}, \bar{s})$,
$\mathbb{B} = \{\texttt{this} \mapsto r, \bar{x} \mapsto \bar{v}\}$, and $\Delta(C, m) = \bar{T} \to \texttt{void}$

SCall $(\mathbb{S}, \mathbb{H}, V, G, C.m(\bar{v})) \to (\mathbb{S}, \mathbb{H}, (\mathbb{B})^C_m :: V, G, \bar{s})$
when $\text{mbody}(C, m) = (\bar{x}, \bar{s})$, $\mathbb{B} = \{\bar{x} \mapsto \bar{v}\}$,
and $\Delta(C, m) = \bar{T} \to T$

SCallVoid $(\mathbb{S}, \mathbb{H}, V, G, C.m(\bar{v})) \to$
$(\mathbb{S}, \mathbb{H}, (\mathbb{B})^C_m :: V, (\texttt{return null;}) :: G, \bar{s})$
when $\text{mbody}(C, m) = (\bar{x}, \bar{s})$, $\mathbb{B} = \{\bar{x} \mapsto \bar{v}\}$,
and $\Delta(C, m) = \bar{T} \to \texttt{void}$

Return $(\mathbb{S}, \mathbb{H}, (M)^C_m :: V, G, \texttt{return } v\texttt{;}) \to (\mathbb{S}, \mathbb{H}, V, G, v)$

Promote $(\mathbb{S}, \mathbb{H}, V, G, v\texttt{;}) \to (\mathbb{S}, \mathbb{H}, V, G, v)$

Sequence $(\mathbb{S}, \mathbb{H}, V, G, s_1 s_2 \ldots s_n) \to (\mathbb{S}, \mathbb{H}, V, (s_2 \ldots s_n) :: G, s_1)$

FrmClose $(\mathbb{S}, \mathbb{H}, V, F :: G, v) \to (\mathbb{S}, \mathbb{H}, V, G, F[v])$

No-op $(\mathbb{S}, \mathbb{H}, V, K :: G, \texttt{;}) \to (\mathbb{S}, \mathbb{H}, V, G, K)$

FrmOpen $(\mathbb{S}, \mathbb{H}, V, G, H[e]) \to (\mathbb{S}, \mathbb{H}, V, H :: G, e)$
when e is not a Value and $H \neq [\cdot]$

Fig. 5. Operational semantics (continued)

As shown in rule New, when a new object is allocated, a fresh reference r is created and the instance fields of the object are initialized according to the following definition, where fields(C) is the set of nonstatic fields in class C:

$$\begin{aligned} \text{init}(\texttt{boolean}) &\stackrel{\text{def}}{=} \texttt{false} \\ \text{init}(C) &\stackrel{\text{def}}{=} \texttt{null} \\ \text{initFields}(C) &\stackrel{\text{def}}{=} \{f \mapsto \text{init}(T) \mid f \in \text{fields}(C), \Delta(C,f) = T\} \end{aligned}$$

Method invocation (Call) creates a new method scope with a single block scope containing mappings for the `this` variable and the method formal parameters. The computation proceeds with the code for the method body, which is extracted from the class table using mbody(). For methods with return type `void`, a dummy `return` is pushed onto the frame stack, as shown in CallVoid. The dummy `return` is needed to pop the method scope from the variable stack when the method terminates. The rule Return pops the method scope from the stack. Static methods are handled similarly (rules SCall and SCallVoid) except that the `this` variable is not available.

The rules Sequence, FrmClose, No-Op, and FrmOpen show how the control of the program is propagated. If a sequence of statements $s_1 s_2 \ldots s_n$ is to be evaluated, the sequence $s_2 \ldots s_n$ is pushed onto the frame stack and s_1 is made active (rule Sequence). Once an expression has been reduced to a value v, the program continues by popping off the topmost frame F from the frame stack and filling its hole with v (rule FrmClose). Similarly, when a statement has been evaluated to a no-op '`;`', the next closed frame on the stack becomes active (rule No-op). Lastly, if the expression currently being evaluated can be decomposed further into a subexpression e and a nontrivial open frame H, the subexpression e is evaluated and H is pushed onto the frame stack (rule FrmOpen).

2.3 Initial and final states

Given a program $prog = cd_1 \ldots cd_n\,\texttt{;}\,s$, the initial state is $\sigma_0 \stackrel{\text{def}}{=} (\mathbb{S}_0, \emptyset, \cdot, \cdot, s)$ where $\mathbb{S}_0$ is the static heap mapping static fields to their initial values:

$$\forall f \in \text{sfields(C)}.\ \mathbb{S}_0(C,f) = \text{init}(T) \quad \text{where } \Delta_f(C,f) = T$$

The function sfields(C) returns the set of static fields defined in the class C.

To correspond with Java, the initial statement s is defined to be $C\texttt{.main();}$, where C is the (unique) class containing a method declaration of the form `static void main()`.[3]

A state is *terminal* if it is of the form $(\mathbb{S}, \mathbb{H}, \cdot, \cdot, \texttt{null})$ or if is of the form $(\mathbb{S}, \mathbb{H}, M, G, e_{exn})$, where e_{exn} is an exception state. Because MJ does not model Java's exception handling features, an MJ program can get "stuck" when a Java program would throw an exception. If e_{exn} is either $\texttt{null.}f\texttt{ = }v\texttt{;}$

[3] Java allows command-line arguments to be passed to main; they have been omitted here for simplicity.

or `null`.$m(\bar{e})$, Java throws `NullPointerException`. When e_{exn} is `(`C`)` r and $\mathbb{H}(r) = (D, \mathbb{F})$ for some $D \not\prec C$, Java throws `ClassCastException`. These terminal configurations are the only cases where a well-typed MJ program can fail to make progress. Previous work on MJ [5] has proved this soundness result using the standard subject reduction technique of Wright and Felleisen [16].

3 Monitoring: events and conditions

This section defines a semantics for Java-MaC safety properties based on MJ's operational semantics. This semantics is based on the earlier Java-MaC work [12,13,11], except that it is considerably more detailed.

The grammar below presents the (mutually recursive) syntax for Java-MaC events E, primitive conditions p, and conditions c.

$$
\begin{array}{rcl}
E & ::= & \texttt{startM(}C.m\texttt{)} \mid \texttt{endM(}C.m\texttt{)} \mid \texttt{update(}C.f\texttt{)} \mid \texttt{update(}C.m.x\texttt{)} \\
& \mid & \texttt{start(}c\texttt{)} \mid \texttt{end(}c\texttt{)} \mid E \texttt{ \&\& } E \mid E \texttt{ || } E \mid E \texttt{ when } c \\
p & ::= & C.m.x \mid C.f \mid p.f \mid p \texttt{ == } p \mid p \texttt{ instanceof } C \\
& \mid & \texttt{true} \mid \texttt{false} \mid \texttt{null} \\
c & ::= & p \mid \texttt{start} \mid \texttt{[}E\texttt{,}E\texttt{)} \mid \texttt{defined(}c\texttt{)} \mid \texttt{inM(}C.m\texttt{)} \\
& \mid & c \texttt{ \&\& } c \mid c \texttt{ || } c \mid c \texttt{ => } c \mid \texttt{!}c
\end{array}
$$

Events are instantaneous incidents that occur during program execution. They take no time and occur during the transition from one state to the next. Java-MaC provides events for monitoring method entry (`startM(`$C.m$`)`), method exit (`endM(`$C.m$`)`), field updates (`update(`$C.f$`)`) and local variable update (`update(`$C.m.x$`)`). Events also can be used to detect when a condition becomes true (`start(`c`)`) or stops being true (`end(`c`)`). More complex events can be composed from the conjunction and disjunction of events, and events may be restricted to occur only when a given condition is true (E `when` c). For example, to detect an update to the field `balance` when the program is inside a method `Bank.deposit`, one would use the Java-MaC event: `update(Bank.balance) when inM(Bank.deposit)`.

Conditions, in contrast to events, are predicates on the program state and thus may hold a particular value for a duration of time. Conditions may denote `true`, `false`, or $\perp$; they are interpreted in a three-valued logic to account for the discrepancy between Java's lexical block structuring and the condition's global scope. A condition has value $\perp$ whenever a variable used to define the condition is not currently in scope.

Primitive conditions are built from Java-like expressions of type `boolean`. One important consequence of formalizing Java-MaC at this level of detail is that it becomes apparent that the primitive conditions permitted should be *pure* (have no side effects on the state) and terminate—these restrictions mean that primitive conditions can be evaluated at any point during the program execution without altering behavior of the system. The one difference from Java is that Java-MaC provides a way to access the state of local variables via the expression $C.m.x$. To handle the new expression form $C.m.x$, the

augmented MJ type system has a judgment to conclude that $\Delta; \Gamma \vdash C.m.x : T$ whenever $T\ x$ `=` v`;` $\in \text{mbody}(C, m)$.

A primitive condition is *permissible* when it has type `boolean` in this augmented type system; formally, p is permissible whenever $\Delta; \emptyset \vdash p$: `boolean`.

Conditions, c, are built from primitive conditions through the usual Boolean operators (and, or, not, and implication), and a few other constructors. Condition `start` holds only at the initial state of the program. The interval condition `[`E_1`,`E_2`)` becomes `true` when event E_1 occurs and becomes `false` when E_2 occurs later. The condition `defined(`c`)` is `true` when the condition c does not evaluate to $\perp$ and `false` otherwise. Finally, the condition `inM(`$C.m$`)` is `true` when the program is executing statements found in the method $C.m$.

3.1 Formal semantics of events and conditions

Having defined an accurate operational model of Java, it is now possible to ascribe meaning to Java-Mac safety policies. Figure 6 gives the formal semantics for Java-MaC events. The judgments are of the form $\sigma \rightarrow \sigma' \models E$, which can be read as "the program in state σ transitions to state σ' and generates event E." As these rules show, events are interpreted to occur during the transition from one state to another. Because the operational semantics of MJ has been designed to closely model Java, defining the Java-MaC events is relatively straightforward. An important question is how events treat subclassing. Should the event `update(`$C.f$`)` be triggered when an object of class $D \prec C$ has its f field updated? The semantics in Figure 6 does respect Java's subclasses, but previous Java-MaC descriptions are ambiguous on this point.

Figure 7 gives the formal semantics for Java-MaC conditions. The judgments are of the form $\sigma \models c = v$ where v is one of `true`, `false`, or $\perp$. Condition `start` holds only for the initial program state σ_0 (rule Start). Interval conditions are more interesting: The initial state satisfies no interval condition because no events have occurred yet (IvlBase). When $\sigma \rightarrow \sigma'$ and σ generates E_1 and σ' does not generate E_2, the interval condition `[`E_1`,`E_2`)` is satisfied. Intervals depend on the history of the computation, so their semantics are defined inductively, as shown in rule IvlInd. This semantics does not lead immediately to an efficient implementation of conditions; rather, these rules are a specification against which an implementation can be verified. Inductive rules lead to a natural proof structure for that verification.

The `defined(`c`)` condition evaluates to `true` when c is not $\perp$, and `false` otherwise (rule Defined). A state satisfies `inM(`$C.m$`)` when there is a method scope for m currently at the top of the variable stack (rules InM1–InM3).

Logical connectives like `&&` are interpreted over three-value logic in the standard way. The expression $v \wedge v'$ is `true` if both v and v' are `true`, `false` if one of v or v' is `false`, and $\perp$ otherwise. Similarly, $\neg\perp$ is $\perp$. Disjunction and implication are defined by their logical encodings into conjunction and negation. These definitions are found in rules And, Not, Or, and Implies.

$$\textsc{StartM}\quad \frac{\mathbb{H}(r) = (D, \mathbb{F}) \quad D \prec C}{(\mathbb{S}, \mathbb{H}, V, G, r.m(\bar{v})) \rightarrow \sigma \models \texttt{startM}(C.m)}$$

$$\textsc{SStartM}\quad \frac{D \prec C}{(\mathbb{S}, \mathbb{H}, V, G, D.m(\bar{v})) \rightarrow \sigma \models \texttt{startM}(C.m)}$$

$$\textsc{EndM}\quad \frac{D \prec C}{(\mathbb{S}, \mathbb{H}, (M)^D_m :: V, G, \texttt{return } v\texttt{;}) \rightarrow \sigma \models \texttt{endM}(C.m)}$$

$$\textsc{FldUpd}\quad \frac{\mathbb{H}(r) = (D, \mathbb{F}) \quad D \prec C}{(\mathbb{S}, \mathbb{H}, V, G, r.f \texttt{ = } v\texttt{;}) \rightarrow \sigma \models \texttt{update}(C.f)}$$

$$\textsc{SFldUpd}\quad \frac{D \prec C}{(\mathbb{S}, \mathbb{H}, V, G, D.f \texttt{ = } v\texttt{;}) \rightarrow \sigma \models \texttt{update}(C.f)}$$

$$\textsc{VarDecl}\quad \frac{D \prec C}{(\mathbb{S}, \mathbb{H}, (M)^D_m :: V, G, T\ x \texttt{ = } v\texttt{;}) \rightarrow \sigma \models \texttt{update}(C.m.x)}$$

$$\textsc{VarUpd}\quad \frac{D \prec C}{(\mathbb{S}, \mathbb{H}, (M)^D_m :: V, G, x \texttt{ = } v\texttt{;}) \rightarrow \sigma \models \texttt{update}(C.m.x)}$$

$$\textsc{StartC}\quad \frac{\sigma \models c = v \quad v \in \{\texttt{false}, \bot\} \quad \sigma' \models c = \texttt{true}}{\sigma \rightarrow \sigma' \models \texttt{start}(c)}$$

$$\textsc{EndC}\quad \frac{\sigma \models c = \texttt{true} \quad \sigma' \models c = v \quad v \in \{\texttt{false}, \bot\}}{\sigma \rightarrow \sigma' \models \texttt{end}(c)}$$

$$\textsc{When}\quad \frac{\sigma \rightarrow \sigma' \models E \quad \sigma' \models c = \texttt{true}}{\sigma \rightarrow \sigma' \models E \texttt{ when } c}$$

$$\textsc{EOr}\quad \frac{\sigma \rightarrow \sigma' \models E_1}{\sigma \rightarrow \sigma' \models E_1 \texttt{ || } E_2} \qquad \frac{\sigma \rightarrow \sigma' \models E_2}{\sigma \rightarrow \sigma' \models E_1 \texttt{ || } E_2}$$

$$\textsc{EAnd}\quad \frac{\sigma \rightarrow \sigma' \models E_1 \quad \sigma \rightarrow \sigma' \models E_2}{\sigma \rightarrow \sigma' \models E_1 \texttt{ \&\& } E_2}$$

Fig. 6. Semantics of Java-MaC events

It remains to give a semantics to the primitive conditions—their interpretation is shown in Figure 8. Because primitive conditions are just Java `boolean` expressions (except for the form $C.m.x$), the idea is to simply evaluate the expressions to either `true` or `false`. To handle the case of $C.m.x$ it is necessary to add an additional operational rule as shown in the figure. The notation $\sigma \rightsquigarrow \sigma'$ denotes a transition step in this augmented semantics. The reflexive, transitive closure of the $\rightsquigarrow$ relation is written $\rightsquigarrow^*$.

Observe that because MJ is sound, if evaluation of a primitive condition halts at a value it must be either `true` or `false`. Because the sublanguage

$$\textsc{Start}\quad \sigma_0 \models \texttt{start} = \texttt{true} \qquad \frac{\sigma \neq \sigma_0}{\sigma \models \texttt{start} = \texttt{false}}$$

$$\textsc{IvlBase}\quad \sigma_0 \models [E_1, E_2) = \texttt{false} \qquad \frac{\sigma' \to \sigma \models E_1 \quad \sigma' \to \sigma \not\models E_2 \quad \sigma' \neq \sigma_0}{\sigma \models [E_1, E_2) = \texttt{true}}$$

$$\textsc{IvlInd}\quad \frac{\sigma' \models [E_1, E_2) = \texttt{true} \quad \sigma' \to \sigma \not\models E_2}{\sigma \models [E_1, E_2) = \texttt{true}} \qquad \frac{\sigma' \to \sigma \models E_2}{\sigma \models [E_1, E_2) = \texttt{false}}$$

$$\textsc{Defined}\quad \frac{\sigma \models c = v \quad v \in \{\texttt{true}, \texttt{false}\}}{\sigma \models \texttt{defined}(c) = \texttt{true}} \qquad \frac{\sigma \models c = \bot}{\sigma \models \texttt{defined}(c) = \texttt{false}}$$

$$\textsc{InM1}\quad \frac{D \prec C}{(\mathbb{S}, \mathbb{H}, (M)^D_m :: V, G, K) \models \texttt{inM}(C.m) = \texttt{true}}$$

$$\textsc{InM2}\quad \frac{D \not\prec C}{(\mathbb{S}, \mathbb{H}, (M)^D_m :: V, G, K) \models \texttt{inM}(C.m) = \texttt{false}}$$

$$\textsc{InM3}\quad (\mathbb{S}, \mathbb{H}, \cdot, G, K) \models \texttt{inM}(C.m) = \texttt{false}$$

$$\textsc{And/not}\quad \frac{\sigma \models c_1 = v_1 \quad \sigma \models c_2 = v_2}{\sigma \models c_1 \texttt{ \&\& } c_2 = (v_1 \wedge v_2)} \qquad \frac{\sigma \models c = v}{\sigma \models \texttt{!}c = \neg v}$$

$$\textsc{Or/Impl}\quad \frac{\sigma \models \texttt{!}(\texttt{!}c_1 \texttt{ \&\& } \texttt{!}c_2) = v}{\sigma \models c_1 \texttt{ || } c_2 = v} \qquad \frac{\sigma \models (\texttt{!}c_1) \texttt{ || } c_2 = v}{\sigma \models c_1 \texttt{ => } c_2 = v}$$

Fig. 7. Semantics of Java-MaC conditions

$$\frac{D \prec C}{(\mathbb{S}, \mathbb{H}, (M)^D_m :: V, G, C.m.x) \rightsquigarrow (\mathbb{S}, \mathbb{H}, (M)^D_m :: V, G, x)} \qquad \frac{\sigma \to \sigma'}{\sigma \rightsquigarrow \sigma'}$$

$$\frac{(\mathbb{S}, \mathbb{H}, V, \cdot, p) \rightsquigarrow^* (\mathbb{S}, \mathbb{H}, V, \cdot, v) \quad v \in \{\texttt{true}, \texttt{false}\}}{(\mathbb{S}, \mathbb{H}, V, G, K) \models p = v}$$

$$\frac{\neg\exists v.(\mathbb{S}, \mathbb{H}, V, \cdot, p) \rightsquigarrow^* (\mathbb{S}, \mathbb{H}, V, \cdot, v) \text{ and } v \in \{\texttt{true}, \texttt{false}\}}{(\mathbb{S}, \mathbb{H}, V, G, K) \models p = \bot}$$

Fig. 8. Semantics of Java-MaC primitive conditions

of primitive conditions terminates in all cases, if evaluation of the primitive condition fails to produce a value, it must have gotten stuck either by trying to interpret a local variable out of scope or by reaching an exception state. In these latter cases, the meaning of the primitive condition is $\bot$.

4 Program instrumentation

Given a set of events and conditions making up a safety property for a program, the Java-MaC system inserts event-monitoring instructions into the

$$\text{IProg}\quad \frac{\bar{cd} \;\Rightarrow \bar{cd}' \qquad C;\texttt{main};\Delta;\emptyset \vdash \bar{s} \;\Rightarrow \bar{s}'}{\bar{cd};\bar{s} \;\Rightarrow \bar{cd}';\; \bar{s}'}$$

$$\text{IClass}\quad \frac{C;\Delta \vdash \bar{md} \;\Rightarrow \bar{md}'}{\texttt{class}\;\; C\;\texttt{extends}\;\; D\;\texttt{\{}\bar{fd}\;\; \bar{md}\texttt{\}} \;\Rightarrow\; \texttt{class}\;\; C\;\texttt{extends}\;\; D\;\texttt{\{}\bar{fd}\;\; \bar{md}'\texttt{\}}}$$

$$\text{IMeth1}\quad \frac{C;m;\Delta;\bar{x}{:}\bar{T},\texttt{this}{:}C \vdash \bar{s} \;\Rightarrow \bar{s}'}{C;\Delta \vdash T\; m\texttt{(}\bar{T}\;\bar{x}\texttt{)\{}\bar{s}\texttt{\}} \;\Rightarrow\; T\; m\texttt{(}\bar{T}\;\bar{x}\texttt{)\{emit(startM(}C.m,\bar{x}\texttt{));}\bar{s}'\texttt{\}}}$$

$$\text{IMeth2}\quad \frac{C;m;\Delta;\bar{x}{:}\bar{T},\texttt{this}{:}C \vdash \bar{s} \;\Rightarrow \bar{s}'}{\begin{array}{l} C;\Delta \vdash \texttt{void}\; m\texttt{(}\bar{T}\;\bar{x}\texttt{)\{}\bar{s}\texttt{\}} \;\Rightarrow \\ \texttt{void}\; m\texttt{(}\bar{T}\;\bar{x}\texttt{)\{emit(startM(}C.m,\bar{x}\texttt{));}\bar{s}'\texttt{;emit(endM(}C.m\texttt{));\}} \end{array}}$$

Fig. 9. Program, class, and method instrumentation

bytecode. This section gives an example of how our formal framework can be used to formalize and reason about such program transformations. Rather than instrumenting bytecode, the scheme presented here instruments Java source programs—establishing a correctness result for the full bytecode transformation is left to future work. In practice, Java-MaC performs a number of optimizations such as monitoring only fields that affect the evaluation of the safety property. The formal version presented here takes a conservative stance by fully instrumenting the program. Doing so simplifies the correctness proofs because they don't have to take into account optimizations; a more realistic model of Java-MaC instrumentation would take optimizations into account.

The instrumented code uses an additional language construct `emit(`ℓ`);` that is intended to model the transmission of an event ℓ to the Java-MaC runtime monitor. In practice, the `emit` instruction would be implemented as an ordinary Java method call to the Java-MaC run-time libraries, but here its behavior is axiomatized by the following operational and typechecking rules:

$$(\mathbb{S},\mathbb{H},V,G,\texttt{emit(}\ell\texttt{);}) \xrightarrow{\ell} (\mathbb{S},\mathbb{H},V,G,\texttt{;}) \qquad \Delta;\Gamma \vdash \texttt{emit(}\ell\texttt{);} : \texttt{void}$$

This rule makes the MJ operational semantics into a *labeled transition system*, where the event labels ℓ are uninterpreted constants. The instrumentation process inserts `emit` instructions into the MJ program as appropriate. Instrumentation correctness (see Section 4.2) requires that when a Java-MaC safety property is interpreted with respect to the labeled transition system, it yields the same outcome as the semantics given in Section 3.

4.1 Formal translation

The instrumentation uses four kinds of transformation rules. The first three are presented on Fig. 9; the rules of the fourth kind are in Fig. 10. Rule IProg instruments a given program $prog = \bar{cd};\bar{s}$. This rule simply instruments all the classes and the initial command vector; it assumes that class C

$$\textsc{IBlk}\quad \frac{C; m; \Delta; \Gamma \vdash \bar{s} \;\Rightarrow \bar{s}'}{C; m; \Delta; \Gamma \vdash \texttt{\{}\bar{s}\texttt{\}} \;\Rightarrow \texttt{\{}\bar{s}'\texttt{\}}}$$

$$\textsc{IFld}\quad \frac{\Delta; \Gamma \vdash e' : T' \qquad C; m; \Delta; \Gamma, x : T' \vdash \bar{s} \;\Rightarrow \bar{s}' \qquad x \notin \mathrm{dom}(\Gamma)}{\begin{array}{l} C; m; \Delta; \Gamma \vdash e.f \texttt{ = } e' \texttt{; } \bar{s} \;\Rightarrow \\ T' \; x \texttt{ = } e' \texttt{; } e.f \texttt{ = } x \texttt{; emit(update(}C.f, x\texttt{)); } \bar{s}' \end{array}}$$

$$\textsc{ISFld}\quad \frac{\Delta; \Gamma \vdash e' : T' \qquad C; m; \Delta; \Gamma, x : T' \vdash \bar{s} \;\Rightarrow \bar{s}' \qquad x \notin \mathrm{dom}(\Gamma)}{\begin{array}{l} C; m; \Delta; \Gamma \vdash C.f \texttt{ = } e' \texttt{;; } \bar{s} \;\Rightarrow \\ T' \; x \texttt{ = } e' \texttt{; } C.f \texttt{ = } x \texttt{; emit(update(}C.f, x\texttt{)); } \bar{s}' \end{array}}$$

$$\textsc{IVar}\quad \frac{C; m; \Delta; \Gamma, x' : T, x : T \vdash \bar{s} \;\Rightarrow \bar{s}' \qquad x, x' \notin \mathrm{dom}(\Gamma)}{\begin{array}{l} C; m; \Delta; \Gamma \vdash T \; x \texttt{ = } e' \texttt{; } \bar{s} \;\Rightarrow \\ T \; x' \texttt{ = } e' \texttt{; } T \; x \texttt{ = } x' \texttt{; emit(update(}C.m.x, x'\texttt{)); } \bar{s}' \end{array}}$$

$$\textsc{IUpd}\quad \frac{\Delta; \Gamma \vdash x : T \qquad C; m; \Delta; \Gamma, x' : T \vdash \bar{s} \;\Rightarrow \bar{s}' \qquad x' \notin \mathrm{dom}(\Gamma)}{\begin{array}{l} C; m; \Delta; \Gamma \vdash x \texttt{ = } e' \texttt{; } \bar{s} \;\Rightarrow \\ T \; x' \texttt{ = } e' \texttt{; } x \texttt{ = } x' \texttt{; emit(update(}C.m.x, x'\texttt{)); } \bar{s}' \end{array}}$$

$$\textsc{IRet}\quad \frac{\Delta; \Gamma \vdash e : T \qquad C; m; \Delta; \Gamma, x : T \vdash \bar{s} \;\Rightarrow \bar{s}' \qquad x \notin \mathrm{dom}(\Gamma)}{\begin{array}{l} C; m; \Delta; \Gamma, \texttt{return } e\texttt{; } \bar{s} \;\Rightarrow \\ T \; x \texttt{ = } e\texttt{; emit(endM(}C.m; x\texttt{)); return } x\texttt{; } \bar{s}' \end{array}}$$

$$\textsc{IIf}\quad \frac{C; m; \Delta; \Gamma \vdash \bar{s}_i \;\Rightarrow \bar{s}'_i \qquad i \in \{1, 2, 3\}}{\begin{array}{l} C; m; \Delta; \Gamma \vdash \texttt{if (}e\texttt{) \{}\bar{s}_1\texttt{\} else \{}\bar{s}_2\texttt{\}; } \bar{s}_3 \;\Rightarrow \\ \texttt{if (}e\texttt{) \{}\bar{s}'_1\texttt{\} else \{}\bar{s}'_2\texttt{\}; } \bar{s}'_3 \end{array}}$$

$$\textsc{INo-op}\quad \frac{C; m; \Delta; \Gamma \vdash \bar{s} \;\Rightarrow \bar{s}'}{C; m; \Delta; \Gamma \vdash \texttt{; } \bar{s} \;\Rightarrow \texttt{; } \bar{s}'} \qquad \frac{C; m; \Delta; \Gamma \vdash \bar{s} \;\Rightarrow \bar{s}'}{C; m; \Delta; \Gamma \vdash pe\texttt{; } \bar{s} \;\Rightarrow pe\texttt{; } \bar{s}'}$$

Fig. 10. Statement instrumentation

contains the `main` method. Rule IClass instruments a class by translating all of its method bodies. The class name is propagated into the translation context for the methods. Instrumenting a method body (IMeth1 and IMeth2) requires that the body of the method itself be instrumented and that the `emit(startM(`$C.m$`));` event be added at the start of the method body. If the method has `void` return type, the corresponding end event must also be added (non-`void` methods are instrumented at their `return` sites). Translation of static methods (omitted from the figure) are the same except that no `this` variable is available in the typing context.

Figure 10 gives the translation of vectors of MJ statements. The judgments are of the form $C; m; \Delta; \Gamma \vdash \bar{s} \;\Rightarrow \bar{s}'$, which says that statements $\bar{s}$ are instrumented to produce statements $\bar{s}'$. The *translation context* consists of C, m, Δ, Γ, where C is the current class, m is the current method name,

$$\begin{array}{lll}
\textsc{StartM'} & \sigma \xrightarrow{\ell} \sigma' \models_t \texttt{startM}(C.m) & \text{when } \ell = \texttt{startM}(C.m, \bar{v}) \\
\textsc{EndM'} & \sigma \xrightarrow{\ell} \sigma' \models_t \texttt{endM}(C.m) & \text{when } \ell = \texttt{endM}(C.m, v) \\
\textsc{FldUpd'} & \sigma \xrightarrow{\ell} \sigma' \models_t \texttt{update}(C.f) & \text{when } \ell = \texttt{update}(C.f, v) \\
\textsc{VarUpd'} & \sigma \xrightarrow{\ell} \sigma' \models_t \texttt{update}(C.m.x) & \text{when } \ell = \texttt{update}(C.m.x, v)
\end{array}$$

Fig. 11. Labeled transition system interpretation of events

Δ is the class table, and Γ is the current type environment. The initial type environment for a method body contains the types of the vector of parameters and the special variable `this` that gives a reference to the object for which the method is being executed.

The translation is straightforward. Instrumenting field updates (IFld & ISFld), local variable updates (IVar & IUpd), and returns (IRet) requires adding some code to store the update value in a temporary variable, performing the source operation, and then emit the appropriate event. Note that naively duplicating the update expression with effects could cause the instrumented program to compute an incorrect answer, which is why the temporary variable is needed. The rest of the rules apply the translation recursively.

The code below gives a method in class C before and after instrumentation.

```
boolean m(boolean b) {
  this.f = b || this.g;
  return this.f;
}
```

```
boolean m(boolean b) {
  emit(startM(C.m,b));
  T x1 = b || this.g;
  this.f = x1;
  emit(update(C.f,x1));
  T x2 = this.f;
  emit(endM(C.m,x2));
  return x2;
}
```

It is easy to prove the following static correctness theorem by induction on the translation derivation; the next section addresses dynamic correctness.

Theorem 4.1 (Static correctness) *Suppose a program* $prog = \bar{cd}; \bar{s}$ *is well typed and* $\bar{cd}; \bar{s} \Rightarrow \bar{cd}'; \bar{s}'$. *Then* $prog' = \bar{cd}'; \bar{s}'$ *is well typed.*

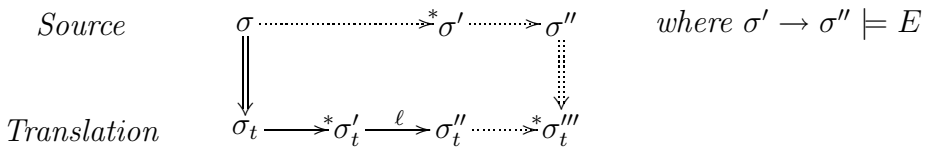
4.2 Instrumentation correctness

This section briefly sketches how to prove that instrumented programs agree with the Java-MaC semantics for uninstrumented programs. The first step is to give a new interpretation of Java-MaC events in terms of the labeled transition system, as shown in the rules of Figure 11—the $\models_t$ symbol indicates the target language interpretation. The semantics for the events not mentioned in the figure remains the same.

The next step of the dynamic correctness proof is to show that any event in labeled transition system correctly simulates a corresponding event in the source-level semantics. Observe that the translation relation $\Rightarrow$ on programs

induces a translation $\sigma \Rightarrow \sigma_t$ on machine states: the translation of the static and dynamic heaps is the identity, the translation of the method scope stack permits additional local variables, and the translation of frames is derived from the translation of statements. The key lemma is the following:

Lemma 4.2 (Event simulation) *Let σ be any well-typed source state and suppose that $\sigma \Rightarrow \sigma_t$. If there exists a label ℓ such that $\sigma_t \rightarrow^* \sigma'_t \xrightarrow{\ell} \sigma''_t$ and $\sigma'_t \xrightarrow{\ell} \sigma''_t \models_t E$, then there exist source states σ' and σ'' and a target state σ'''_t such that $\sigma \rightarrow^* \sigma'$, $\sigma' \rightarrow \sigma'' \models E$ and $\sigma''_t \rightarrow^* \sigma'''_t$ and $\sigma'' \Rightarrow \sigma'''_t$. Pictorially: (the solid parts of the figure imply the existence of the dotted parts)*



5 Related work

Besides the previous research on the Java-MaC framework itself [12,13,11], this work is related to a number of research areas, including formalizing languages, reference monitors, and profiling.

The operational semantics presented here builds on the work on Middleweight Java [5], which is itself a variant of Featherweight Java (FJ) [10]. Other variants of FJ include different features that might be interesting to incorporate into the model of Java-MaC. For example, the original work on FJ treats both exceptions and inner class. Similarly Banerjee and Naumann [2] treat mutable state, private fields and class-based visibility.

A second related area is reference monitoring and software fault isolation (SFI) [15]. Schneider [14] defines a class of security policies specified by security automata that work via execution monitoring. The goal of his paper is to characterize the subset of policies enforceable by run-time monitoring. SASI (Security Automata SFI Implementation) [6] is an implementation of Schneider's security automata. It enforces security policies by modifying object code for a target system before that system is executed. Instead of instrumenting the code to ensure that all memory accesses and control transfers are safe as in SFI, SASI inserts code that simulates a specified security automaton and halts the target system if the automaton rejects its input. Evans and Twyman [7] propose a similar system for instrumenting programs to enforce security policies, but their policies lack a formal semantics. Walker et al. [4,3] extend Schneider's work by considering enforcement mechanisms that can do more than simply terminate faulty programs.

There is some similarity between monitoring and profiling. To measure performance, common tools such as `gprof` modify code to collect statistics such as amount of time spent in functions. A different class of profiling tools, such as `strace` or DCPI [1], works without requiring program recompilation.

6 Discussion and future work

This paper has presented a formal semantics for Java-MaC that should serve as the basis for future research. Building the semantics highlighted some potential design tradeoffs and questions about the Java-MaC system. Currently, the implementation does not support use of the `instanceof` operator in primitive conditions, which would be useful for writing accurate safety predicates. Similarly, unlike the semantics proposed here, condition specifications do not respect Java subclasses. These decisions in the implementation result from a design that permits programs to be monitored remotely, which makes network communication and remote references issues for efficiency and correctness. Another difficulty with remote monitoring is handling objects and references, because the monitor must essentially reconstruct the memory layout of the target program. The semantics here treats the events and conditions as being evaluated locally. It may be possible to reconcile the two views by taking advantage of Java's support for inheritance and dynamic dispatch.

One appealing future direction is to investigate ways to make the safety policies specifiable in Java-MaC more expressive. For example, one could contemplate adding new events that distinguish between constructor and ordinary method calls, or new events to describe exceptions, garbage collection, or block entry and exit. Conditions that monitor memory consumption might also be a useful addition to Java-MaC.

Another important future direction is to formulate a lightweight version of Java bytecode and formalize MJ compilation. Doing so would allow the correctness of the Java-MaC byte-code instrumentation to be verified with respect to the MJ semantics presented here. Such a result would involve a more elaborate simulation proof than the one given here. A promising step in this direction is work by Freund and Mitchell on formalizing Java bytecode [8].

6.1 Acknowledgments

Many thanks to Insup Lee and Oleg Sokolsky for helping us better understand the Java-MaC framework.

References

[1] J. Anderson, W. Weihl, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, and C. Waldspurger. Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[2] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[3] L. Bauer, J. Ligatti, and D. Walker. A calculus for composing security policies. Technical Report TR-655-02, Princeton University, 2002.

[4] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security*, July 2002.

[5] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. Mj: An imperative core calculus for java and java with effects. Technical Report 563, University of Cambridge, April 2003.

[6] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, September 1999.

[7] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, May 1999.

[8] S. N. Freund and J. C. Mitchell. A type system for the java bytecode langauge and verifier. *Journal of Automated Reasoning*, 2003. (to appear).

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.

[10] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java. In *Conference of Object-Oriented Programming, Systems, Languages and Applications*, volume 34 of *ACM SIGPLAN Notices*. ACM Press, October 1999.

[11] M. Kim. *Information Extraction for Run-time Formal Analysis*. PhD thesis, University of Pennsylvania, 2001.

[12] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java programs. In *1st International Workshop on Run-time Verification*, July 2001.

[13] I. Lee, H. Ben-Abdallah, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Monitoring and checking framework for run-time correctness assurance. In *Proceedings of the 1998 Korea-U.S. Technical Conference on Strategic Technologies*, 1998.

[14] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[15] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating System Principles (SOSP)*, pages 203–216. ACM Press, December 1993.

[16] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. Preliminary version in Rice TR 91-160.