# Self-Stabilizing Java

Yong hun Eom and Brian Demsky
University of California, Irvine
{yeom,bdemsky}@uci.edu

## Abstract

Self-stabilizing programs automatically recover from state corruption caused by software bugs and other sources to reach the correct state. A number of applications are inherently self-stabilizing—such programs typically overwrite all non-constant data with new input data. We present a type system and static analyses that together check whether a program is self-stabilizing. We combine this with a code generation strategy that ensures that a program continues executing long enough to self-stabilize. Our experience using SJava indicates that (1) SJava annotations are easy to write once one understands a program and (2) SJava successfully checked that several benchmarks were self-stabilizing.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability

***General Terms*** Languages, Reliability

***Keywords*** Self-Stabilization, Software Robustness

## 1. Introduction

Software bugs have long plagued software systems. Proving software systems correct remains a difficult problem. Practitioners have instead relied on extensive testing to verify that programs operate correctly in the scenarios that they are likely to be used (i.e., their comfort zones [14]). Despite extensive testing effort, it is common for unusual inputs to trigger a bug that corrupts the program's state. After a bug corrupts a program's state, the program can in general behave arbitrarily. Self-stabilizing systems, however, are guaranteed to reach the correct state after a finite number of steps [6].

This paper presents a combination of a type system and static analyses that together check that a software application is self-stabilizing with respect to rarely-triggered software bugs and certain types of transient hardware failures. The self-stabilization property checked by SJava is very powerful—it ensures that if a user returns to using a software system in the ways it was tested for, the software system will resume working correctly.

### 1.1 Basic Approach

SJava checks that program executions eventually transition from incorrect states to the correct state by showing that incorrect values eventually leave the execution and are replaced by correct values. The approach targets programs that have a main event loop that acquires new inputs at each iteration. SJava partitions both the heap and variable memory locations into abstract locations using location types. Location types form a lattice. SJava checks two properties that together ensure that a program self-stabilizes. The first property is that values only flow from higher to lower abstract locations (hence referred to as the *flow-down rule*). Like information flow [12], SJava must enforce the flow constraint on both the explicit flows caused by assignments and the implicit flows caused by branching on a value and then storing a value. SJava leverages a linear type system to prohibit aliases that could potentially subvert the flow-down rule. The second property is that values can only remain at a location for a bounded time.

Figure 1 presents a graphical depiction of the effect of these two properties on an execution whose state is corrupted by a bug. The red ×'s indicate corrupted values while the green ✓'s indicate correct values. These two properties together ensure that after a bounded time, memory locations with the highest location types have the same value in both the correct execution and the buggy execution. As the execution progresses, memory locations with lower location types have their corrupted values overwritten with correct values. Eventually, all memory locations have correct values and the buggy execution has self-stabilized into the correct state.

### 1.2 Error Model

SJava checks that applications self-stabilize in response to errors that occur inside the event loop. We make the assumption that the application successfully reaches the entrance to the event loop. We believe that this is a reasonable assumption; a human can often intervene for systems that fail to start up. Furthermore, we assume that all input reads are performed unconditionally in every iteration of the event loop, to eliminate the possibility of framing errors.

SJava primarily targets rarely-triggered software bugs. We assume that code is mostly correct with the possibility that rare input sequences may cause the program to behave incorrectly. We model these errors as incorrect state transitions to an incorrect state. The SJava system guarantees that if a program in a bad state is fed a

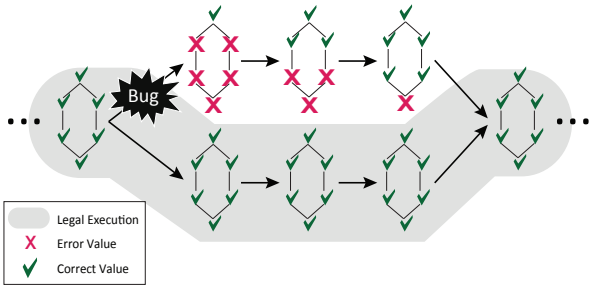


**Figure 1.** Trace of the Program State



*PLDI'12,* June 11–16, 2012, Beijing, China.

previously tested input sequence longer than the self-stabilization period, the program will reach the exact state as in the test.

Our basic approach is also applicable to certain types of hardware errors. For example, SJava can guarantee self-stabilization for hardware errors that create faulty inputs to an application. In general, SJava can handle transient hardware errors if the hardware errors (1) do not affect the termination of loops, (2) do not corrupt event loop invariant values in variables or memory, (3) do not violate type safety, and (4) do not cause the execution to jump to arbitrary statements. Compiler and hardware implementations can ensure these types of guarantees thus preserving the self-stabilization property [7–9].

### 1.3 Usage Scenarios

We envision several usage scenarios focusing on embedded controllers and stream decoders. We describe a few scenarios in more detail below:

- **Multimedia Streaming:** Unexpected values can easily cause video and audio decoders to crash or misbehave and prevent playing the remainder of a multimedia stream. Self-stabilizing decoders might fail to decode short periods of a stream due to software bugs, but these failures will only be transient and the remainder of the stream will be correctly decoded.
- **Embedded Controllers:** Many embedded controllers are intended to operate for long periods without human intervention. Software bugs can cause these controllers to enter states where they fail to perform as intended. In some cases, it may take significant time for humans to recognize that the controller is misbehaving and reset the software systems. Self-stabilizing controllers are guaranteed to return to correct operation.
- **Safety Critical Code:** A concern with safety critical systems is that possibly undetected bugs might transition a software system to a corrupted state that prevents further operation. While self-stabilizing systems do not guarantee the absence of bugs, they bound the time frame that even undetected errors can affect the correct operation of a system. Self-stabilization is not intended to replace the rigorous validation processes that are currently used to ensure correctness, but rather to complement these processes to limit the consequences of bugs that inevitably slip through.

### 1.4 Contributions

This paper makes the following contributions:

- **Basic Approach:** It presents a basic approach for checking whether programs are self-stabilizing.
- **Types for Self-Stabilization:** It presents a type system that ensures that values eventually flow out of a program to return the program to the correct state.
- **Implementation:** It presents an implementation of an SJava compiler including the type system and static analyses.
- **Experience:** It presents our experiences using SJava to check that several applications are self-stabilizing.

The remainder of the paper is organized as follows. Section 2 presents an example. Section 3 presents the location type abstraction. Section 4 presents the type checking rules. Section 5 presents the static analysis that checks that corrupted values are eventually evicted. Section 6 presents our approach for ensuring that event loop iterations terminate. Section 7 presents our approach to code generation. Section 8 overviews the basic correctness argument. Section 9 evaluates our approach on several benchmark applications. Section 10 presents related work; we conclude in Section 11.

```
1  @LATTICE("DIR<TMP,TMP<BIN")
2  public class WDSensor{
3    @LOC("BIN") private WindRec windRec = new WindRec();
4    @LOC("DIR") private int dir;
5
6    @LATTICE("STR<WDOBJ,WDOBJ<IN")
7    @THISLOC("WDOBJ")
8    public void windDirection(){
9      SJAVA:
10     while(true){  // main event loop
11       @LOC("IN") int inDir = Device.readSensor();
12       // move old wind directions one step down
13       windRec.dir2 = windRec.dir1;
14       windRec.dir1 = windRec.dir0;
15       // add a new wind direction
16       windRec.dir0 = inDir;
17       @LOC("STR") String strDir = calculate();
18       broadcastChange(strDir);
19     }
20   }
21
22   @LATTICE("OUT<CAOBJ")
23   @THISLOC("CAOBJ")
24   @RETURNLOC("OUT")
25   public String calculate(){
26      @LOC("CAOBJ,TMP") int majorDir;
27      @LOC("OUT") String strDir;
28      // calculate the majority
29      ...
30      this.dir = majorDir;
31      strDir = convertToString(majorDir);
32      return strDir;
33   }
34 }
35
36 @LATTICE("DIR2<DIR1,DIR1<DIR0")
37 class WindRec{
38   @LOC("DIR0") public int dir0;
39   @LOC("DIR1") public int dir1;
40   @LOC("DIR2") public int dir2;
41 }
```

**Figure 2.** Wind Direction Sensor Example

## 2. Example

We present a Java weather station example to illustrate SJava. The weather station processes sensor inputs and broadcasts them. Figure 2 presents the main event loop for the weather station. The loop reads sensor data from a device in Line 11, adds it to the `WindRec` object that stores the most recent three directions in Line 16, and determines the wind direction by calling the `calculate()` method. To compensate for sensor errors, the `calculate()` method examines the previous three directions and computes the median to discard invalid direction values.

### 2.1 Self-Stabilization

Software bugs or hardware failures can corrupt an application's state, causing problems during its subsequent execution. In our example, there are many possible ways of arriving at an incorrect state. In one possible scenario, the device returns an erroneous value, which is not one of the 16 possible directions. This value is stored in and corrupts the `windRec` data structure. Next, the `convertToString()` method throws an uncaught null pointer exception due to the corrupted value in `windRec`. Finally, the program fails to show the current wind direction.

Once a bug occurs, the system potentially has undefined and possibly undesirable behavior because the program's state may contain corrupted values. Our observation is that the system will resume normal behavior if the execution eventually arrives at the correct state. In the example, after any erroneous value enters the `windRec` data structure the program would return to the correct execution after at most three iterations of the main loop. SJava only

computes whether a program self-stabilizes and not when it self-stabilizes. However, it is possible to compute bounds by analyzing the lattice.

SJava analyzes programs that employ the main event loop pattern: an outer loop retrieves a new input, processes it, and produces the output. The goal is to check that all effects of a value disappear after a finite number of main event loop iterations.

To show that the program cannot remain indefinitely in incorrect states, we first establish an ordering relation on the memory locations in the program. The ordering relation forms a location hierarchy that constrains how values flow through the program. Precisely, it ensures that values flow in one direction: from higher locations to lower locations. Therefore, assignments are only allowed if the location of the left-hand side of an assignment is lower than the location of the value to be stored.

However, forbidding assignments that violate ordering constraints is not sufficient to obtain the desired property because it does not force values in the program's memory to be evicted within a finite number of steps. Therefore, the SJava compiler ensures that non-loop invariant values that are live are evicted within one loop iteration. Our analysis is designed to check eviction of memory locations within one loop iteration as checking longer eviction times is unlikely to provide significant benefits but requires more sophisticated techniques that can reason about predicates.

Even if a program does not store any values indefinitely, it may crash before it reaches a legal state. Given the guarantee that the system will return to the correct state, the developer may choose to have the program log and then ignore uncaught exceptions. SJava optionally supports ignoring uncaught exceptions to ensure that the program will execute long enough to self-stabilize.

### 2.2 SJava Annotations

Line 1 of Figure 2 defines the location hierarchy for the fields in the `WDSensor` class. The ordering relation, as defined by the < operator, allows values to flow from the location type that appears after the operator to the location type that appears before the operator. The example location hierarchy constrains values to only flow down from locations with the `BIN` location type to locations with the `TMP` location type to locations with the `DIR` location type (flows directly from the `BIN` location to the `DIR` location that skip the `TMP` location are allowed). We use the annotation `@LOC` to declare that the field `windRec` has the location type `BIN` in Line 3.

Line 22 defines the location hierarchy for the `calculate()` method. Local variables have composite locations, where a composite location consists of a location in the method's hierarchy followed by any number of locations from field hierarchies. The ordering of composite locations is given by comparing the elements of the two composite locations in lexical order. Line 26 declares the composite location type `LOC`$\langle$`CAOBJ,TMP`$\rangle$ for the `majorDir` variable. This location is lower than $\langle$`CAOBJ,BIN`$\rangle$ and higher than $\langle$`CAOBJ,DIR`$\rangle$ since the field hierarchy has the ordering relations `DIR` $\sqsubset$ `TMP` and `TMP` $\sqsubset$ `BIN`. Developers must assign a location in the hierarchy to the `this` variable using the annotation `@THISLOC`. Line 23 assigns the `CAOBJ` location to the `this` variable.

The special label `SJAVA` in Line 9 specifies that the `while` loop in the next line is the main event loop. We next discuss how the compiler checks that the program statements evaluated through the iteration of the main event loop do not violate the flow constraints.

### 2.3 Checking Self-Stabilization

The SJava compiler checks the parts of the program that are callable from the main event loop. First, it checks that every field, variable, and parameter accessed in the main event loop has been annotated with a location type. Next, it checks that all assignments respect the ordering relation. Specifically, an assignment is only allowed if the left-hand side's location is lower than the value being assigned. For example, the assignment to `this.str` in line 30 is valid because the location type $\langle$`CAOBJ,TMP`$\rangle$ of the source value is higher than the location type $\langle$`CAOBJ,DIR`$\rangle$ of the destination.

For every call site, the compiler must check that value flows created by the callee do not violate the caller's ordering constraints. Location lattices in SJava are not global—each method instance has its own locally-scoped location lattice. SJava's method local location lattices allow a single method to be used in several different contexts in which the arguments do not have the same location types. It also makes our type system composable—the SJava location type system captures the behavior of a method and not how the method happens to be used in the overall system. SJava must ensure that the location lattice of the callee enforces the flow constraints of the caller's argument. Alternatively, this check can be viewed as verifying that the caller and callee lattices can be merged into a single combined lattice. Specifically, the compiler (1) checks that the ordering constraints of the arguments in the caller satisfy the ordering constraints required by the location types of the callee's parameters and (2) computes the highest caller location for the return value that is consistent with the callee's ordering constraints. For example, the compiler processes the location annotations for the `calculate` method to determine that its return value is lower than its receiver object, and then checks that this is consistent with the caller's lattice (i.e., that the variable `strDir` has a lower location than the `this` variable).

The SJava compiler also checks that all memory locations accessed by the event loop are either loop invariant or were overwritten in either the current or previous loop iteration. In the example, all variables and fields are obviously overwritten by each iteration of the event loop. A termination analysis (Section 6) ensures that each iteration of the event loop terminates by prohibiting recursive calls [1] and checking that inner loops terminate.

## 3. Location Type System

In SJava, every memory location has a location type in addition to its Java type. A location type constrains which types can be stored in the corresponding memory location. The compiler checks that every assignment moves values from memory locations with higher location types to memory locations with lower location types.

### 3.1 Location Types

A program execution may create a statically unbounded number of concrete memory locations. We therefore map the concrete memory locations to a finite set of location types. Location types are assigned by the developer to field declarations, variable declarations, and parameter declarations.

### 3.2 Location Type Lattice

The location hierarchy is defined by the lattice $\langle L_{SET}, \sqsubseteq \rangle$, where $L_{SET}$ is the set of location types and the binary relation $\sqsubseteq$ establishes an ordering between location types. It is useful to note that we use both the reflexive partial ordering ($\sqsubseteq$) and the corresponding strict partial ordering ($\sqsubset$). We make use of the reflexive partial ordering to support the standard lattice machinery while our type checking rules rely on the strict partial ordering.

For example, `low` $\sqsubset$ `high` means that the location `high` is higher than `low`, specifying that values can legally flow from memory locations with the location type `high` to memory locations with the location type `low`. The location lattice includes the top and bottom locations. The top location $\top$ is the highest location, whose values

[1] Note that the type system and other static analyses currently handle recursive calls. The restriction against recursive calls is only due to limitations in our termination analysis.

| Annotation | Role | Applied to |
|---|---|---|
| @LATTICE | Defines a location hierarchy | Classes and Methods |
| @METHODDEFAULT | Defines the class-wide default method hierarchy | Classes |
| @LOC | Assigns a location to a declaration | Fields, Variables, and Parameters |
| @THISLOC | Selects a location for the 'this' reference | Methods |
| @PCLOC | Selects a location for the program counter | Methods |
| @GLOBALLOC | Selects a location for static references | Methods |
| @RETURNLOC | Selects a location for a return value | Methods |
| @DELEGATE | Transfers ownership | Method Parameters |
| @TRUST | Indicates that a method was manually inspected | Methods |

**Figure 3.** Annotations

can flow anywhere. The bottom location $\perp$ is the lowest location, any value can flow to such locations. The location lattice has the *meet* operator $\sqcap$, which computes the greatest lower bound (GLB) of any two location types in the lattice. Our GLB operation is the standard lexicographic GLB.

### 3.3 Method and Field Location Lattices

SJava has a separate location hierarchy for each class and method. Each class has a field hierarchy lattice that defines an ordering between fields of the same object instance. Each method has a method hierarchy lattice that is used to establish an ordering between the different variables in the method. Both the field and method hierarchies are defined as lattices. The next section discusses how the elements of method and field hierarchies are combined into a composite location that orders all memory locations in a program.

### 3.4 Composite Location Types

A composite location type is a sequence of location elements—the first element of the composite location is a method location from the current method's method hierarchy lattice, followed by a sequence of zero or more field locations. Consider the field access expression `foo.bar.z`. It has the composite location type $\langle$`FOO`, `Foo.BAR`, `Bar.Z`$\rangle$, where the local variable `foo` has the location type `FOO`, the field `bar` has the location type `BAR` in the field hierarchy of class `Foo`, and `z` has the location type `Z` in the field hierarchy of class `Bar`.

For every field access, the compiler computes the composite location that describes the position in the ordering of the field by combining the composite location type of the reference variable with the field location element. Developers have the option to declare any level of the composite location for local variables. This enables a developer to set the local variable's ordering relative to specific fields so that a local variable with a composite location can take a value from one field, and then store it back to another field in the same object.

#### 3.4.1 Comparison

The comparison of two composite locations is based on lexicographical ordering of the location elements. The comparison begins with the first elements of the two composite locations. If the first elements are not identical, then the lattice for the first location determines the ordering relation of two locations. If the first elements are identical, the comparison continues onward to later elements in the composite location types.

The composite location with n location elements has a set of partial orders $\{\sqsubseteq_1, \sqsubseteq_2, ..., \sqsubseteq_n\}$. The partial ordering relation of the composite location is defined as follows:

$$\langle a_1, a_2, ..., a_n\rangle \sqsubseteq_C \langle b_1, b_2, ..., b_n\rangle \Leftrightarrow$$
$$\exists j \in \{1, ..., n\}.(a_j \sqsubset_j b_j \vee (j = n \wedge a_j = b_j)) \wedge \forall i < j.a_i = b_i$$

Location elements at position *i* come from a lattice that defines a partial ordering relation $\sqsubseteq_i$. If two field elements are from different classes, then the composite location types are incomparable.

Lexicographical ordering addresses the following issue with implicit information flows through heap paths. Consider a heap path to a primitive field (e.g., `x.f`, where `f` is a primitive field)—if a value is high enough to flow to a reference along the path to the object with field (e.g., the variable `x`), with lexicographical ordering it is also high enough to legally flow to the field (e.g., `f`). The ordering therefore simplifies the typing rules because such flows cannot violate the ordering relation.

### 3.5 Inheritance

As a subclass inherits fields and methods from its parent class, it must preserve the ordering hierarchy from the parent class. The compiler checks that every location defined in the parent is included in the subclass's field hierarchy. The subclass can of course declare new locations in its hierarchy. The compiler checks that the value flows allowed by the subclass are the same in the parent to prevent the ordering constraints from being subverted by an overridden method or a cast. Checking the hierarchy of an overridden method is exactly the same as the field hierarchy check with the additional constraint that the parameters must have the same declared locations.

### 3.6 Location Type Annotations

SJava's location type annotations are written using standard Java annotations. Figure 3 summarizes the basic types of SJava annotations. The annotation `@LATTICE` defines a location hierarchy and can be applied to both class and method declarations. Figure 4 presents the grammar for lattice declarations and location declarations. The value in the `@LATTICE` annotation consists of a series of binary relation entries that define the ordering relation. The binary relation uses the inequality notation <, `x < y` means that a value can flow from `y` to `x`.

*latticeDecl* := `@LATTICE` ( *orderDecls*,*sharedLocDecls* )
*orderDecls* := *orderDecls*, *orderDecl* | *orderDecl*
*orderDecl* := *location* < *location*
*sharedLocDecls* := *sharedLocDecls*, *location** | *location**
*compositeLoc* := `@LOC` ( *locationList* )
*deltaLoc* := `@DELTA` ( *locationList* | *deltaLoc* )
*locationList* := *locationList*, *locElement* | *locElement*
*locElement* := *location* | ClassName.*location*

**Figure 4.** Location Declaration and Annotation Grammar

Every method must have a method hierarchy, but declaring a lattice for every method can be labor intensive. Therefore, the SJava provides a default lattice for the method. The `@METHODDEFAULT` annotation on the class declaration defines a class-wide method lattice. If a method is not annotated with a method hierarchy using the `@LATTICE` annotation, the method uses the default lattice for the class. When many methods' behaviors are similar, the default lattice can significantly reduce the annotation burden.

The developer specifies the location types of variable, field, and parameter declarations using the annotation `@LOC` followed by a parenthesized composite location. The `@THISLOC` annotation designates a location in the method hierarchy for the `this` variable. This allows the compiler to derive the proper composite location for a value accessed through the `this` variable and compute its ordering relations relative to other local variables.

Assigning a location to the static field references is done in a similar manner to the `this` variable. The `@GLOBALLOC` annotation specifies the location of static fields in the method lattice. The type checker ensures that method lattices consistently order globals relative to arguments using the checks that we use to preserve ordering between arguments. At this point, SJava does not support defining an ordering relation between static fields from different classes. Instead, we envision that static fields will be primarily used to store constants, and therefore can be assigned to a very high location. However, static fields could be supported by partitioning them into groups, using annotations to describe the locations of groups, and checking that different methods use these locations consistently.

The program counter location tracks implicit flows. If a method can be safely called when the program counter location is lower than one of the parameter locations, the developer can use the `@PCLOC` to declare the initial location for the program counter. Otherwise, the program counter has the top location.

As noted, developers can assign any composite location to local variables and parameters. For example, on Line 26 from Figure 2, the annotation `@LOC("CAOBJ,TMP") int majorDir` indicates that the variable `majorDir` has the composite location type $\langle$`CAOBJ`, `TMP`$\rangle$. For method declarations, the annotation `@LOC` specifies the location type for a parameter and the annotation `@RETURNLOC` specifies the location type of the return value.

## 4. Flow-down Rule

SJava's type checking is independent from the standard Java type checking, so this section will focus only on the location type checking rules. In SJava, every memory location has a location type that captures how values can flow into and out of that memory location.

We define the following notations: The symbol $L$ represents a composite location type, which is a sequence of location elements. The symbol $l$ represents a location element. The elements of a composite location are $\langle l_0, l_1, ..., l_{n-1}\rangle$. The function *size(L)* returns the size of the sequence representation of the location type $L$. The static environment $\Gamma$ provides a mapping from identifiers to location types, $\Gamma_m$ provides a mapping for the callee $m$. The *callee_init_env* function returns the initial type environment at the beginning of the method body. The notation $\Gamma(x)$ gives a mapping from the identifier $x$ to either the location type $L$ or the location element $l$ bound to the field. One of the identifiers is the *program counter location* `pc` that represents the current context constraint that restricts the location type of the destination of any assignments. The purpose of the *program counter location* is to track implicit value flows.

### 4.1 Location Type Checking Rules

Figure 5 presents the type checking rules. There are two kinds of type judgment rules. The judgment $\Gamma \vdash e : L$ states that the expression $e$ has the location type $L$ in the $\Gamma$ environment. The judgment $\Gamma \vdash e$ states that the expression $e$ is well-typed with respect to the environment $\Gamma$. The notation $\Gamma[pc = v]$ represents the same environment except that the program counter *pc* is bound to new value $v$. We next describe the basic checking rules:

$$\text{(LITERAL)}\ \frac{\mathit{true}}{\Gamma \vdash \mathit{literal} : \top} \qquad \text{(OP)}\ \frac{\Gamma \vdash e_0 : L_0 \quad \Gamma \vdash e_1 : L_1 \quad L = L_0 \sqcap L_1}{\Gamma \vdash e_0 \square e_1 : L}$$

$$\text{(ASSIGN)}\ \frac{\Gamma(x) = L \quad \Gamma \vdash e : L_e \quad \Gamma \vdash L \sqsubset L_e \quad \Gamma \vdash L \sqsubset \Gamma(pc)}{\Gamma \vdash x = e : L}$$

$$\text{(VAR)}\ \frac{\Gamma(x) = L}{\Gamma \vdash x : L} \qquad \text{(FD\_R)}\ \frac{\Gamma \vdash e : L_e \quad \Gamma(f) = l_f \quad L = L_e \oplus l_f}{\Gamma \vdash e.f : L}$$

$$\text{(FD\_W)}\ \frac{\Gamma \vdash e_0 : L_0 \quad \Gamma(f) = l_f \quad \Gamma \vdash e_1 : L_1 \quad L = L_0 \oplus l_f \quad \Gamma \vdash L \sqsubset L_1 \quad \Gamma \vdash L \sqsubset \Gamma(pc)}{\Gamma \vdash e_0.f = e_1 : L}$$

$$\text{(IF)}\ \frac{\Gamma \vdash c : L_c \quad \Gamma[pc = \Gamma(pc) \sqcap L_c] \vdash e_{i \in \{1,2\}}}{\Gamma \vdash \texttt{if}(c)\ e_1\ \texttt{else}\ e_2}$$

$$\text{(WHILE)}\ \frac{\Gamma \vdash c : L_c \quad \Gamma[pc = \Gamma(pc) \sqcap L_c] \vdash e}{\Gamma \vdash \texttt{while}(c)\ e}$$

$$\text{(ARRAY\_VAR)}\ \frac{\Gamma \vdash a : L_a \quad \Gamma \vdash i : L_i \quad L = L_a \sqcap L_i}{\Gamma \vdash a[i] : L}$$

$$\text{(ARRAY\_ASG)}\ \frac{\Gamma \vdash a : L_a \quad \Gamma \vdash i : L_i \quad \Gamma \vdash e : L_e \quad \Gamma \vdash L_a \sqsubset L_i \quad \Gamma \vdash L_a \sqsubset L_e \quad \Gamma \vdash L_a \sqsubset \Gamma(pc)}{\Gamma \vdash a[i] = e : L_a}$$

$$\text{(CALL\_SITE)}\ \frac{\begin{array}{c}\Gamma_m = \mathit{callee\_init\_env}(\texttt{@RETURNLOC}(V_{rv})\ \texttt{@THISLOC}(V_t)\ \texttt{@PCLOC}(V_{pc})\\ m(\texttt{@LOC}(V_{p1})p_1, ..., \texttt{@LOC}(V_{pn})p_n))\\ \Gamma_m \vdash p_0 : L_0^m \ ...\ \Gamma_m \vdash p_n : L_n^m \qquad \Gamma \vdash a_0 : L_0\ ...\ \Gamma \vdash a_n : L_n\\ \forall i, j \in \{0, ..., n\}\ \Gamma_m \vdash L_i^m \sqsubset L_j^m \ \Rightarrow\ \Gamma \vdash L_i \sqsubset L_j\\ X_i = (\texttt{if}\ (\texttt{sub}(L_i^m, 0, 1) = L_0^m)\ \texttt{then}\ L_0 \oplus \texttt{sub}(L_i^m, 1, \mathit{size}(L_i^m))\ \texttt{else}\ \bot)\\ \forall i \in \{1, ..., n\}\ \Gamma \vdash (X_i \sqsubset L_i \vee X_i = L_i)\\ I_r = \{i \mid \forall i \in \{0, ..., n\},\ \Gamma_m \vdash (L_{rv}^m \sqsubset L_i^m \vee L_{rv}^m = L_i^m)\} \quad L_r = \bigsqcap_{i \in I_r} L_i\\ \forall i \in \{0, ..., n\} \quad L_i^m \sqsubset \Gamma_m(pc) \ \Rightarrow\ L_i \sqsubset \Gamma(pc)\\ \text{call is virtual w/ multiple targets} \ \Rightarrow\ \Gamma_m(pc) \sqsubset L_0^m\end{array}}{\Gamma \vdash a_0.m(a_1, ..., a_n) : L_r}$$

$$\langle l_0, l_1, ..., l_n\rangle \oplus l_m = \langle l_0, l_1, ..., l_n, l_m\rangle$$

$$\texttt{sub}(\langle l_0, l_1, ..., l_n\rangle, i, j) = \langle l_i, ..., l_{j-1}\rangle$$

**Figure 5.** Location Type Checking Rules

**Literal:** Every literal value has the highest location type `TOP` in the location hierarchy, denoted by $\top$. Therefore, all constant values in a program can flow to any memory location.

**Operation:** The operation rule derives the location type of an arithmetic expression of the form $e_0 \square e_1$. The derived location type is the greatest lower bound of the location types of two operands.

**Variable Assignment:** A variable assignment causes a value flow from its right-hand side to its left-hand side. The `ASSIGN` rule checks that the destination's location type is lower than the source's location type. The last premise tracks implicit flows by checking the context constraints due to control flow.

**Field Read:** For a field read expression `e.f`, a new composite location is derived by appending the location type of the field `f` to the location type of the base expression `e`. The binary operation $\oplus$ adds a new location element to the end of the composite location.

**Field Assignments:** The field assignment rule is similar that for variable assignments except that the composite location type of the left-hand side is derived from the field access expression.

### 4.2 Arrays

The naïve approach to handling arrays is to assign all the elements of an array to the same location type. This approach prohibits value flows between elements of the same array and is therefore too restrictive for most real-world applications.

SJava supports two different approaches to arrays. In the first approach, the array have a special *shared location* type that allows value flows between array elements provided that the entire array is cleared out (or lowered) at the same time at some point in each iteration of the event loop. Section 4.7 presents more details on shared locations.

Alternatively, SJava can assign unique locations to each array element. In this case, the array elements are ordered in sequence with the first element having the lowest location and the last having the highest. The SJava library then provides an `insert` method that shifts all the elements down by one index and assigns a new value to the last position. The type system assumes that this method moves all values in the array one step down. The eviction analysis ensures that the insert method is called at least once in each loop iteration.

To ensure that a value flow between an index value and an array does not violate ordering constraints, SJava has two separate rules for array expressions. The `ARRAY_VAR` rule checks that an array access expression has a location type that is the greatest lower bound of both the location type of the array and the location type of the index. The array assignment rule `ARRAY_ASG` has to consider the relative ordering of an array and the index value used for that array. The location type of the array should be lower than the location type of the index since the value of the array index affects how values flow into the array. The rule also checks that the location type of an array variable is lower than the value expression being assigned.

### 4.3 Implicit Flow

Conditional branches may cause implicit value flows that could violate the ordering constraints. The example code below introduces an implicit flow. The value of the variable `a` in the `if` condition statement affects the value assigned to the variable `b`. As a result, if the location type of the variable `b` is higher than the location type of the variable `a`, it is a violation of the ordering constraint since there exists a value flow between them.

```
if (a>0) b=1; else b=0;
```

To prevent implicit flows that violate ordering constraints, the `IF` and `WHILE` rules update the program counter location with the location type of the `if` condition or `while` condition, respectively. This ensures that any conditional assignments in the body of the `if` statement or loop prohibit implicit flows that are not permitted.

For the example, after evaluating an `if` statement, the `pc` is set to the location type of the condition expression $\langle A \rangle$, then the compiler checks that the left-hand side of the assignment `b` has a location type lower than `pc`'s location type.

The compiler also ensures that a method call in a conditional branch respects implicit flows. The callee's program counter location reflects the location type of the call site's context constraint.

### 4.4 Method Invocation

Location type annotations in method declarations impose restrictions that both the caller and the callee must respect. When arguments are passed to the parameters of the call site, the caller must respect the callee's restrictions on the relative orderings of the arguments and the return value. The callee must in turn respect the constraints its declared interface places on its internal value flows. The two sets of restrictions together guarantee that method invocation respects the ordering constraints of the caller and the callee. Alternatively, the call site checks can be viewed as checking that the collection of method lattices can be transformed into one global method lattice that is consistent with the program's value flows.

#### 4.4.1 Call Site Checking

The parameter's location type describes how the location type of an argument in the caller is transferred into the method hierarchy

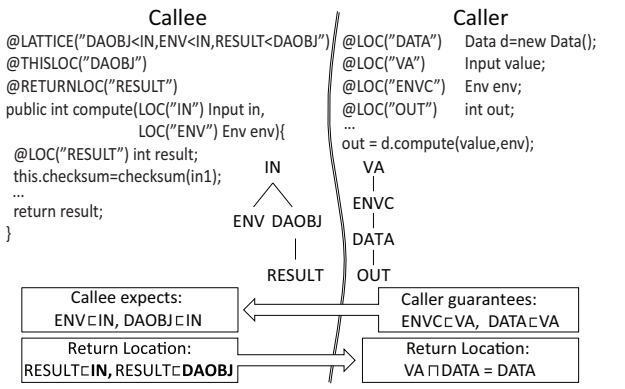


**Figure 6.** Method Invocation

of the callee. From the perspective of the callee, relative orderings between parameters establish ordering constraints on the location type of a value passed in, which the caller must respect when it assigns values to arguments.

Type checking the call site ensures that the caller provides arguments that respect the callee's ordering constraints, which requires the type checker to check the mapping of location types from the caller's arguments to the callee's parameters. For each call site $m_s(a_0, a_1, ..., a_n)$ to the corresponding method declaration $m(p_0, p_1, .., p_n)$, the rule `CALL_SITE` checks that for any two parameters $p_i$ and $p_j$ if the callee has the ordering relation $p_i \sqsubset p_j$ between parameters, then the caller has to have the corresponding ordering relation $a_i \sqsubset a_j$ between its arguments. If the callee does not have any ordering relation between two parameters, the caller does not need to respect any ordering constraints on the corresponding two arguments because it implies that the callee will not have a value flow between two parameters. Figure 6 illustrates how the compiler checks constraints of the caller and the callee. The callee has two ordering relations among parameters, `ENV` $\sqsubset$ `IN` and `DAOBJ` $\sqsubset$ `IN` in its hierarchy, and therefore the callee imposes two ordering constraints on the caller's arguments. The caller must guarantee that the corresponding arguments have same relation in its hierarchy, in this case `ENVC` $\sqsubset$ `VA` and `DATA` $\sqsubset$ `VA`.

If the first element of a parameter location type matches the location type of the current object `this` and a field element is in the next position, the callee establishes ordering constraints relative to the field lattice of the current object. This provides more specific constraints on the ordering relations of fields in the current object; the caller needs to satisfy constraints on not only the ordering relations between arguments, but also the ordering relations of fields given by the field hierarchy of the object referenced type of `this`. For example, suppose that the parameter has the location type $\langle$`IOBJ`, `F`$\rangle$ and the receiver object has the location type `IOBJ`. The corresponding argument in the caller is required to be higher than or equal to $\langle$`0`, `F`$\rangle$ if the object whose method is being invoked has the location type $\langle$`0`$\rangle$ in the caller.

#### 4.4.2 Return Value Location

The call site rule uses the location types of the parameters and the return value to conservatively compute the set of flows that the callee may produce. The rule then computes a caller location type that allows all of these flows in the callee context. This check can be viewed as checking that it is possible to combine the callee and the caller lattices in a way that allows all existing information flows.

The `CALL_SITE` rule in Figure 5 computes the caller's return value location as follows. First, it computes the set of parameter location types that are higher than or equal to the declared return lo-

cation type. Parameters that are not higher than the return location type are irrelevant because the ordering constraints prevent value flows from these parameters. In Figure 6, the `compute` method provides a set of parameter location types {`IN`, `DAOBJ`} for calculating the return value location. Next, the rule creates a set of argument location types in the caller that correspond to this set of parameters and then computes the greatest lower bound of the location types of these arguments. The caller in Figure 6 computes the greatest lower bound of `VA` and `DATA` since the callee locations `IN` and `DAOBJ` correspond to the locations `VA` and `DATA`, respectively, in the caller's hierarchy. In this case, the location type of the return value is `DATA`, which means that the caller must not return a value that is lower than the location type `DATA`. If the return value of the method is the right-hand side of an assignment, the rule `ASSIGN` then checks that the return value location is higher than the left-hand side. The last assignment in the right column of the Figure 6 satisfies the ordering constraints with the location type of the return value `DATA`.

### 4.5 Objects

Non-static fields are always accessed through an instance reference variable. The location types of instance reference variables provide a way to compute the relative ordering between other local instances. In the case of copying field values from one instance to another instance, the static checking checks that the location of the source instance is higher than the destination. The current implementation of SJava prohibits recursive data structures. Future work could relax this constraint. One approach is to require programs to delete all references to a recursive data structures within some loop iteration bound.

#### 4.5.1 Aliasing

Aliasing refers to the situation in which multiple references point to the same object. In SJava, if two aliased references to the same object were allowed to have different location types, this would open the possibility of values flowing from the lower locations to higher locations through the aliased references in violation of the flow-down rule. For example, suppose that a program creates two references with different location types to the same object. The static checking as described would allow the program to use the higher reference to access a value from a field that was written to using the lower reference. One approach to ensuring that aliasing is safe is to ensure that all aliases to an object have the same location type.

SJava uses linear types to restrict aliasing. SJava's linear type system prohibits multiple heap aliases from referencing the same object. This implies that the heap that can be updated by the event loop in SJava must be a forest of objects (multiple disjoint trees). SJava allows limited aliasing from local variables and parameters—variable aliases are allowed as long as all aliases have the same location type. A side effect is that if an alias exists to any object in a tree, the location type of all objects in that tree cannot be changed.

#### 4.5.2 Ownership Transfer

In some cases, a method may need to lower the location type of an object passed in as a parameter. SJava supports ownership transfer to allow a caller method to transfer ownership of a non-aliased reference to a callee method.

The method acquires ownership of a non-aliased reference through parameters with the `@DELEGATE` annotation. Exclusive ownership allows the method to lower the location type of a reference, to transfer its ownership to other methods, to create heap references to the object, and to remove subtrees from the heap reachable from the object. Ownership transfer guarantees that a given reference must be owned by only one method and no aliases to the object exist in other scopes. In this respect, the caller has the responsibility to pass a unique reference argument to the callee. Static checking of the caller checks that all references to the object are dead after the call site. Methods can only returned owned references. Returning aliased references could be supported with an annotation that declares that the return value is aliased and gives the parameter from which the alias was obtained.

The ownership status of a variable can be either (1) a parent method owns the tree (i.e. the object is aliased), (2) in the case of temporary variable used to traverse a locally own tree, the local variable that contains the owned reference to the tree's root, or (3) the current local variable owns the reference. We only allow changing the level of a reference if (1) the reference owns the tree and (2) no other local variables refer to objects in the same tree. We allow transferring ownership of a component object of a tree only if (1) the current method owns the tree and (2) the temporary variable used to remove the reference is the only reference other than the owning reference.

### 4.6 Delta Locations

Code often uses temporary variables to access data structures. Including location types for all of these temporary variables would greatly complicate both the method and field hierarchies. Instead, SJava provides a special function `delta` that takes a composite location and generates a new composite location, called a *delta location*, which is lower than the input composite location and higher than everything that is lower than the input composite location. The delta function is applied to a whole composite location. The delta function can be applied to the output of itself to generate a descending series of composite locations.

Consider a code segment that copies a value from an object field to a local variable, computes a value using the local variable, and then stores the value to a different, lower field of the same object. This value flow involves a variable, so it requires the variable to have a location between the two fields in the field hierarchy. With the delta function applied to the composite location of the source field, it is straightforward to generate a composite location for the local variable that is lower than the source field and higher than the destination field. In the example from Section 2, the composite location ⟨`CAOBJ`, `TMP`⟩ can be replaced with `delta`(⟨`CAOBJ`, `BIN`⟩) in Line 26. It generates a new location that is lower than ⟨`CAOBJ`, `BIN`⟩ and higher than all locations below ⟨`CAOBJ`, `BIN`⟩.

For correctness purposes, delta functions are syntactic sugar that introduces new elements into the hierarchy of the last component of the composite location and updates the lattice appropriately.

### 4.7 Shared Locations

The flow-down rule ensures that every assignment lowers the location type of the value being assigned. This constraint prohibits common computations that read from a set of memory locations, perform computation, and store the results into the same set of memory locations. This constraint also prohibits simple `for` loops, e.g., `for(int i=0;i<10;i++) ;`, as the increment operation violates the standard flow-down rule.

SJava provides shared location types for primitive types to allow developers to specify a set of memory locations with the same composite location that values can flow freely between. The developer can assign the same shared location to multiple locations, and then freely flow values between those locations. Shared location are explicitly listed in the lattice annotation with a *.

SJava must ensure that a program actually clears out all memory locations with the same shared location type and does not merely shuffle corrupted values between memory locations. A shared memory location is cleared when a value from a higher location is written and remains cleared until the program overwrites that memory location with a value with the same shared location type.

SJava checks that all memory locations with the same shared location type are simultaneously in the cleared state at least once per an event loop iteration (or before every use). Therefore, the program cannot use a shared location to store values indefinitely (and circumvent self-stabilization) even though a shared location may keep values through assignments. In the next section, we describe how SJava uses static analysis to check this constraint. Among other uses, shared locations are useful for allowing index variables in `for` loops.

## 5. Eviction of Values

Although the flow-down rule ensures that all value flows respect the ordering constraints, it does not ensure that values leave a memory location in a bounded time period. For example, suppose that a variable is written by one event loop iteration, and all future iterations of the event loop read that value. In this scenario, a corrupted value can remain indefinitely and the execution may never self-stabilize. This section presents a static analysis that ensures that a memory location does not store values indefinitely.

### 5.1 Definitely-Written Analysis

The definitely-written analysis ensures that reads inside the event loop either read (1) a value written outside of the event loop or (2) a value written by the current or the immediately preceding event loop iteration. This ensures that corrupted values cannot remain live in the same memory location indefinitely.

The analysis checks that for each memory location $M$ that the event loop either (1) overwrites $M$ or (2) overwrites a reference that lies on the heap path to $M$ (thus lowering M or making it unreachable). The analysis operates in two stages. In the first stage, it computes read and write sets. In the second stage, it checks that the event loop body respects the definitely-written constraints.

#### 5.1.1 Computing Read and Write Sets

The analysis generates the read set $R^m$, the may-write set $OW^m$, and the must-write set $WT^m$ for the main event loop and each method $m$ that is callable from the main event loop. Elements of these sets are represented by heap paths, which are n-tuples of references that describe the sequence of heap accesses to reach a memory location from one of the method's parameters. For example, accessing the field `f` of an expression `x` that is reachable from the parameter $\mathtt{p}_1$ through a sequence of references $r_1, ..., r_n$ generates the heap path $\langle \mathtt{p}_1, r_1, ..., r_n, \mathtt{f}\rangle$. The analysis computes a mapping $HP$ that maps a variable to the heap path that describes the sequence of references from the parameter to the object the variable references. The analysis can safely ignore reads and writes on local variables as they will go out of scope when the method exits.

Our analysis uses a standard fixed-point algorithm. Figure 7 presents the transfer functions for computing read and write set. We define the helper function $HP(\mathtt{x})=\{hp \mid \langle \mathtt{x}, hp\rangle \in HP\}$.

**Read:** The set $R^m$ contains the heap paths that must either be never written in the loop or overwritten before the method $m$ is called in a different iteration of the event loop. The field read statement `x=y.f` generates a new heap path for `x` by appending the field `f` to the heap path $HP(\mathtt{y})$. The read statement also adds the corresponding heap path to $R^m$ if it or a prefix may not have been overwritten since the method entry.

**Write:** The field write statement `x.f=y` adds the heap path through `f` to the set $WT$ and the set $OW^m$. Note that it is not necessary to ensure that we update existing heap paths when creating a new heap path. The reason is that the flow-down rule ensures that reference involved in new heap paths must flow down.

**Call Site:** For the call site $\mathtt{c}(a_0, ..., a_n)$, the callee's read and write effects are propagated to the caller. The analysis first computes the sets $OW^c_{bound}$, $WT^c_{bound}$ and $R^c_{bound}$ for all possible callees.

| st | |
|---|---|
| `x=y.f` | $HP' = HP \cup \{\langle x, p \oplus f\rangle \mid p \in HP(\mathtt{y})\}$<br>$R_{new} = \{p \oplus f \mid p \in HP(\mathtt{y}), \exists p' \in WT \Rightarrow \neg\mathtt{Pre}(p \oplus f, p')\}$<br>$R^{m'} = R^m \cup R_{new}$ |
| `x.f=y` | $WT' = WT \cup \{HP(\mathtt{x}) \oplus f\}$<br>$OW^{m'} = OW^m \cup \{HP(\mathtt{x}) \oplus f\}$ |
| `call` $\mathtt{c}(a_0, ..., a_n)$ | $R^c_{bound} = \bigcup_{c \in calleeSet(c), i \in \{0,...,n\}} \{HP(\mathtt{a_i}) \odot r \mid r \in R^c \wedge \mathtt{Eq}(r, p_i)\}$<br>$OW^c_{bound} = \bigcup_{c \in calleeSet(c), i \in \{0,...,n\}} \{HP(\mathtt{a_i}) \odot r \mid r \in OW^c \wedge \mathtt{Eq}(r, p_i)\}$<br>$WT^c_{bound} = \bigcap_{c \in calleeSet(c), i \in \{0,...,n\}} \{HP(\mathtt{a_i}) \odot r \mid r \in WT^c \wedge \mathtt{Eq}(r, p_i)\}$<br>$R_{new} = \{p \mid p \in R^c_{bound}, \exists p' \in WT \Rightarrow \neg\mathtt{Pre}(p, p')\}$<br>$R^{m'} = R^m \cup R_{new}$<br>$OW^{m'} = OW^m \cup OW^c_{bound}$<br>$WT' = WT \cup WT^c_{bound}$ |
| `merge` | $WT' = \bigcap_{i \in pred(st)} WT_i$ |
| `exit` | $WT^m = \bigcap_{i \in pred(st)} WT_i$ |

**Figure 7.** Transfer Functions for Computing Read and Write Sets

$$\langle a_0, a_1, ..., a_n\rangle \oplus b = \langle a_0, ..., a_n, b\rangle$$
$$\langle a_0, a_1, ..., a_n\rangle \odot \langle b_0, b_1, ..., b_n\rangle = \langle a_0, ..., a_n, b_1, ..., b_n\rangle$$
$$\mathtt{Eq}(\langle a_0, ..., a_n\rangle, \langle b_0, ..., b_n\rangle) = (a_0 = b_0)$$
$$\mathtt{Pre}(\langle a_0, ..., a_n\rangle, \langle b_0, ..., b_k\rangle) = k \le n \wedge (\langle a_0, ..., a_k\rangle = \langle b_0, ..., b_k\rangle)$$

**Figure 8.** Auxiliary Operators and Functions

The operator $\odot$ converts the effects of the callee to the caller by replacing a parameter reference with the corresponding argument heap path. For example, the argument has the heap path $\langle \mathtt{d}, \mathtt{g}\rangle$ that is passed as the parameter `x` to the callee $c$. If the callee has the two read tuples $\langle \mathtt{x}, \mathtt{y}, \mathtt{a}\rangle$ and $\langle \mathtt{x}, \mathtt{y}, \mathtt{b}\rangle$ in its set $R^c$, the corresponding caller context read tuples $\langle \mathtt{d}, \mathtt{g}, \mathtt{y}, \mathtt{a}\rangle$ and $\langle \mathtt{d}, \mathtt{g}, \mathtt{y}, \mathtt{b}\rangle$ are added to the set $R^c_{bound}$. The set $R^c_{bound}$ and $OW^c_{bound}$ are the union of all possible callees, and the set $WT^c_{bound}$ is the intersection of all possible callees. Then, the set $R^m$ of the caller gains an element of $R^c_{bound}$ if that element or some prefix has not been written by the caller $m$. The set $OW^m$ and $WT$ also gain write effects from $OW^c_{bound}$ and $WT^c_{bound}$, respectively.

**Control Flow Join:** The join operation of the must-write set $WT$ is intersection because memory locations must be overwritten on all possible program paths.

**Method Exit:** Without loss of generality, we assume the method exit node appears after all return nodes of the method. The set $WT^m$ is the intersection of the set $WT_i$ for all predecessors $i$.

**Arrays:** Arrays are handled in a similar fashion to fields with special support for the array specific calls in the SJava library.

#### 5.1.2 Checking the Main Event Loop

We next describe the operation of the definitely-written analysis on the event loop. The definitely-written analysis must ensure that all memory locations that the event loop reads are either (1) loop invariant, (2) overwritten in the current loop before the read, or (3) overwritten in every loop iteration. For reads from local variables, we check with a straightforward dataflow analysis that either (1) all

reaching definitions are from outside the event loop, (2) the variable must be overwritten in the current loop before the read statement, or (3) the variable must be overwritten in every loop iteration.

For reads from the heap, we check the condition in the main event loop at each call site and each field dereference. For each newly read heap path $p$ in the set $R_{new}$ for the statement $st$, we check that either:

1. that the heap path $p$ is never written in the event loop, i.e., $p \notin OW$,
2. that the heap path $p$ or some prefix is overwritten in the current event loop before executing the statement $st$, i.e., $\exists p' \in WT_{st}, \texttt{Pre}(p, p')$, or
3. that the heap path $p$ or some prefix is overwritten in every loop iteration, i.e., at every loop backedge statement $st' \exists p' \in WT_{st'}, \texttt{Pre}(p, p')$.

### 5.2 Shared Location Extension

Recall that all memory locations with the same shared location must be overwritten with values from a higher location at the same time. Our analysis for shared locations checks that one of following conditions is satisfied for all memory locations with the same shared location type at the same program point: (1) the value in the memory location was written in the current loop iteration from a higher memory location or (2) one of the references in the heap path that leads to the memory location was written in the current loop iteration.

The shared location analysis is an extension to the definitely-written analysis. The analysis computes a mapping from a shared location to a set of heap paths or variables that belong to the same shared location. When a program statement writes a shared memory location, the analysis adds the memory location to the set only if the value being assigned to is from a higher location. If the value has the same shared location type, the memory location is removed from the set.

Whenever all memory locations with the same shared location type are cleared out, the definitely-written analysis adds the shared location type to the currently cleared shared locations types set for the current program point. The analysis then uses this set to compute at each statement which shared locations must be cleared in the current loop iteration before reaching a given statement. It then uses the shared locations must be cleared set in an analogous fashion to the set $WT$.

## 6. Termination of Event Loop Iterations

SJava ensures that values flow out of a program after a bounded number of iterations of the main event loop. It is of course possible for an iteration of the main event loop to fail to terminate due to memory corruption, a software bug, or by design. In this situation, an application could fail to self-stabilize because it never finishes an iteration of the main event loop and therefore the corrupted values never leave. We must therefore assure that every loop iteration of the main event loop terminates.

The halting problem is of course known to be undecidable. The proposed termination analysis instead targets checking common terminating loop patterns. In the next section, we describe how SJava checks that the execution of inner loops terminates. SJava addresses the possibility of looping recursive calls by prohibiting recursive calls.

### 6.1 Loop Termination Analysis

We implemented a simple loop termination analysis. Our analysis verifies loop termination if a loop both (1) has an index variable and at each iteration the index variable is incremented by a constant value and (2) every iteration of the loop evaluates at least one inequality of the appropriate form for the increment statement and that consists of the index variable and a guard value that does not change over the iterations. The most common type of for-loop follows this pattern.

For every nested loop in the event loop, the compiler first computes the set of induction variables and then checks that every loop iteration evaluates at least one conditional loop exit that is composed of an appropriate inequality of an induction variable and an invariant value. This check guarantees that the loop terminates because at each iteration, the induction variable proceeds toward the termination condition by increasing its value.

### 6.2 Loop Termination Annotations

This simple analysis cannot always determine that a loop terminates. Prohibiting the remaining loops is unlikely to be practical. To support loops where the SJava compiler cannot statically reason about termination, SJava provides two loop annotations: the maximum loop annotation and the unchecked annotation.

The maximum loop annotation modifies the original loop to enforce a developer-specified loop iteration bound. When the compiler flags a possible infinite loop, the developer can simply annotate the loop with a maximum loop annotation to force the loop to terminate within a given iteration bound. The compiler then generates code to enforce this bound.

It can be difficult to specify a maximum iteration bound for certain types of loops. In this case, developers can manually analyze the loop. If the developer manually checks that the loop terminates, the developer can apply a special unchecked annotation to the loop. Unchecked loops are indicated with a Java loop label that starts with the string `TERMINATE_`. The compiler then trusts that the developer has checked that the annotated loop always terminates.

## 7. Code Generation for Self-Stabilization

SJava checks that if an execution continues, it will self-stabilize into the correct state. However, an uncaught exception can cause the program to terminate before it self-stabilizes. There are two different approaches for handling such errors. In many cases, it may be appropriate to simply restart the program. Even in such cases, checking that the program is self-stabilizing ensures that silent software bugs cannot leave the program in incorrect states indefinitely. In other cases, the restart time may be significant. In these cases, the developer may choose to ignore uncaught exceptions. Note that the period of incorrect behavior caused by ignoring the error is limited because SJava ensures that the execution will self-stabilize into the correct state. Our compiler therefore implements an option to eliminate uncaught exceptions—we simply generate code that logs the error and then gives the error cases defined behavior. For example, under this option dereferencing a null pointer simply produces another null pointer. Virtual method calls on null receiver objects pose a related problem—self-stabilization may rely on code inside one of the targets of the call executing. In this case, the execution would choose one of the possible method targets to execute.

## 8. Correctness

We next sketch the basic correctness argument for SJava.

**Lemma 1** (Top Values)**.** *If an SJava program type checks and passes the static analyses, then memory locations with the top location type have the correct values after one loop iteration.*

*Proof Sketch:* After one loop iteration, memory locations with the top location cannot be corrupted as they are either constants or input data for the current loop iteration.

**Lemma 2** (Propagation)**.** *If SJava type checks a program and all memory locations with location types with a maximum distance of n from the top value in the lattice have correct values at the beginning of a non-erroneous loop iteration, then all live memory locations with locations types with a maximum distance of n + 1 from the top value in the lattice must have correct values at the end of the loop iteration.*

*Proof Sketch:* If a memory location has a location type with a maximum distance of $n + 1$, then all memory locations that are higher than it must have correct values (they have lower maximum distances) by assumption. If a value in a memory location is live (there is a read that could read this value before it is overwritten), then SJava requires each loop iteration to overwrite the memory location. The new value must be correct as all locations higher than this memory location have correct values.

**Theorem 1** (Self-Stabilization)**.** *If an SJava program type checks and passes the static analyses, then it self-stabilizes.*

*Proof Sketch:* The location type lattice has a finite height, therefore by induction on Lemmas 1 and 2 all non-constant memory locations will eventually have the correct values.

## 9. Evaluation

We implemented a compiler for SJava and evaluated it by annotating three existing Java applications: JLayer, an MP3 decoder; LEA, an eye-tracker; and Sumo Robot, a robot controller.

We conducted experiments in which we randomly injected errors to measure the self-stabilizing behavior of each benchmark. Our compiler generated error injection code that randomly selects memory and mathematical operations, and replaces the original value with a random value.

### 9.1 MP3 Decoder

JLayer is an MP3 decoder and is available at `http://www.javazoom.net/javalayer/javalayer.html`. MP3 files are composed of a sequence of frames. In JLayer, every event loop iteration retrieves a `BitStream` object that corresponds to a frame. The `BitStream` object reads and returns one frame from the input audio file and maintains persistent state to store the file offset. The `BitStream` object was carefully manually designed to be self-stabilizing by resyncing to MP3 frames, and we annotated the `BitStream` object as trusted to self-stabilize.

We focused our efforts on automatically checking that the more complex decoder is self-stabilizing. The decoder is self-stabilizing because the event loop flushes out all non loop-invariant state within a bounded time. As long as the event loop retrieves new valid audio frames, it will resume the normal behavior from an arbitrary state of the non-loop invariant storage.

We modified the program to minimize interactions between trusted components and checked components. Interactions between trusted and checked components (i.e., more than one method call per loop iteration or inputs to trusted components) make manually checking trusted code more difficult. For example, if a trusted method takes parameters from the self-stabilization code, the developer must reason about the behavior of the trusted code with potentially corrupted parameter values.

We found that the last two steps in the original code, the IMDCT and the Synthesis Filter Bank, use the results from the previous frame. In the original code, the results from two different loop iterations are stored in the same array, which makes it difficult to reason about value flows. Therefore, we use two separate arrays—one to store the merged results and one to forward results from the current loop iteration to the next loop iteration.

Our experiments were designed to qualitatively evaluate how the program self-stabilizes after an error corrupts its state. We randomly injected an error during the program's execution and measured the time until the program resumes outputting the correct values. We performed 1,000 trials of the experiment and observed 466 trials with corrupted outputs. Figure 9 shows the distribution of the number of output samples from when the error is injected until the point at which JLayer returned to outputting normal values. JLayer returned to normal behavior in less than 500 output samples when an error was injected into the transformation to generate PCM samples, which is the final step of the decoding process. The large peak at 1,700 samples occurs when an error is injected into the frequency domain transformations for one of the two granules that comprise a frame. Because the frequency transformation computations involve many operations, such error injections are likely. The corrupted results of the computation then continue to affect the output for approximately 1,700 samples. In general, errors that corrupted an internal data structure, for instance the buffer index of the `BitStream`, affected more output samples. In all cases, errors affected fewer than 2,208 output samples.

Figure 10 shows a section of JLayer's decoded audio signal output from one of the trials. The normal output of JLayer is plotted in blue. The output from the execution with error injection is plotted in red. The red box is due to the signal for the error execution deviating from the normal execution by oscillating at high frequency between -32,767 and 32,767. After 1,630 samples the program behavior returned to normal until termination.

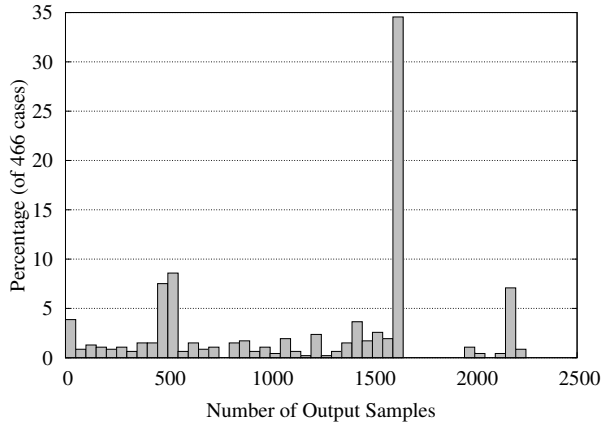


**Figure 9.** Distribution of the number of output samples required for the program to return to the normal behavior after an error injection.

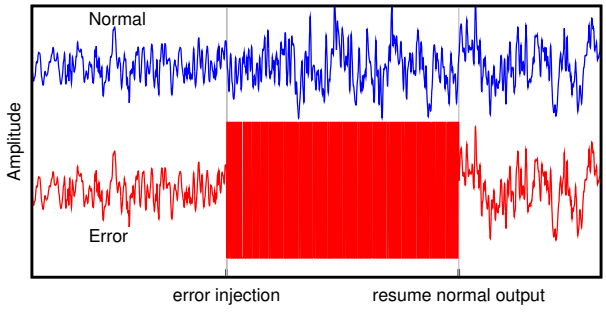


**Figure 10.** MP3Decoder output: normal execution (blue, top) and execution with injected error (red, bottom). After error injection, the signal oscillates rapidly until it resumes normal behavior.

| Benchmark | Location | Lattice | Method Default | Lines |
|---|---|---|---|---|
| MP3 Decoder | 690 | 54 | 24 | 15,634 |
| Eye Tracking | 143 | 33 | 21 | 4,571 |
| Robot Control | 77 | 15 | 13 | 3,201 |

**Figure 11.** Number and Type of Annotations

### 9.2 Eye Tracking

LEA is a lightweight eye tracking algorithm library and is available at `http://sourceforge.net/projects/lea-eyetracking/`. At each iteration, LEA takes an input image from a web cam, tracks eye movements, and returns relative movements in 8 directions (i.e., up, left, down, ...). The algorithm first detects a face in the input image, which allows it to localize the search region for eye detection. When an eye position is detected, LEA determines movement by computing the deviation from the last three eye positions. LEA stores the last three eye positions in the array to provide a better estimation. Every iteration inserts a new position at the beginning of the array and shifts the previous results down by one. All memory locations except the array of previous positions are overwritten in each iteration. The previous positions are not used to compute new positions while the direction result is derived from the three previous positions. Thus, the program returns the correct execution within three iterations of the event loop.

We annotated LEA and checked that it self-stabilizes. We modified the original code to convert multi-threaded code to single-threaded code. No other modifications were necessary to verify self-stabilization.

We performed 100 executions with injected errors and observed 8 executions with changed output samples. For each trial, we randomly injected errors at 10 consecutive instructions and compared eye positions and direction decisions with the correct output generated by the non-instrumented version. In our 8 trials, LEA returned to outputting correct values in the next iteration of the main event loop. While the SJava annotations imply a longer worst-case self-stabilization period, in practice it is hard to trigger this behavior through randomized error injection.

### 9.3 Robot Control

Sumo Robot controls robots and is available at `http://java.net/projects/sumorobots/`. The goal of a Sumo Robot is to push the opponent out of a ring while staying away from the ring edge. A robot is equipped with two types of sensors: a sonar sensor, which detects the opponent, and a line sensor, which detects the ring edge. Each iteration of the event loop reads data from the sensors, selects a movement type and speed, and then generates a motor controller command. The `StrategyMgr` object implements the analysis of the sensor input and the selection of the movement type. Once a motor control command is sent to the hardware, the command persists until another command is sent. We do not attempt to automatically analyze the motor controller because it is not stateless. We annotated the motor controller as trusted code, and modified the original code to make sure that every iteration overwrites the command arguments to the motor controller.

We evaluated the behavior of the Sumo Robot controller using simulated sensor inputs. We performed 100 error-injected executions. For each execution, we recorded the movement decisions of the strategy controller at every iteration of the event loop, and then compared the movement decisions with the output from an error-free execution. In the presence of the injected errors, we observed 54 trials with changed outputs and observed that the Sumo Robot controller resumed the normal behavior in the next iteration of the main event loop after the error occurred.

### 9.4 Annotation Effort

In SJava, the developer must annotate all variable, field, and method declarations accessed by the event loop. We found the location type errors from the compiler helpful in correctly annotating the code.

Figure 11 summarizes our annotation effort. For each benchmark, we list the number of location assignments using `@LOC` in the Location column, the number of lattice definitions using `@LATTICE` in the Lattice column, the number of default method lattice definitions using `@METHODDEFAULT` in the Method Default column, and lines of code including libraries. We found defining the structure of lattices to be straightforward. The default method lattice reduces the number of annotations because many methods share a similar structure and therefore we simply reused the same lattice. While the annotations do require extra developer effort, we found that this effort was small for our benchmarks, especially when compared to the effort of manually checking self-stabilization.

In our experience, SJava required minimal development effort once we understood the overall design of the program because value flows often reflect interactions between individual modules in the design specification. If a developer intends to develop a new software system for SJava, our experience leads us to believe that effort involved in annotating code will marginally exceed the amount of effort required to write Java types.

## 10. Related Work

Self-stabilization was initially suggested by Dijkstra in the context of robust distributed algorithms [6]. There has recently been work by Dolev et al. [7–9] that proposes techniques to ensure that the underlying layers (processor, operating system, and compiler) preserve the self-stabilizing nature of an application. Their work does not check that the actual application is self-stabilizing. Therefore, our work on SJava to check that applications self-stabilize is complementary to this work.

Language-based information flow employs type systems to check that information flows in an application do not violate the desired requirements[12, 15, 16]. Our approach is similar in some aspects. The key differences are that our approach must support much finer-grained divisions of data and must check that values only remain in a given location for a bounded time.

Linear types have been leveraged in programming languages[10, 11, 19] for various purposes including safe concurrent programming, resource management, and protocol checking. SJava uses a modified linear type system to avoid soundness problems from aliasing. Termination analysis plays an important role in verifying safety critical systems.

Even though it is not possible to have a sound and complete termination analysis, several proposed techniques are mature enough to analyze termination in many cases [1–4, 18]. If necessary, we could easily replace our termination analysis with more sophisticated approaches.

Failure-oblivious computing [13] enables programs to continue execution past memory errors by manufacturing values for reads or discarding writes. This approach works well for applications with short error propagation distances. Our work is complementary in that it can guarantee that a program has short error propagation distances. Other work detect bugs and tries re-execution in a slightly different environment [17]. Data structure repair [5] takes an interventional approach; upon detecting data structure corruption, it repairs them with respect to a specification. Data structure repair only guarantees that a program will reach some consistent state, while our work guarantees that all effects of the bug eventually disappear. Moreover, our approach does not require a specification and therefore eliminates the need to precisely define correct behavior.

## 11. Conclusion

Self-stabilization has long been proposed as an approach for fault tolerance. Wide scale guarantees of self-stabilization have the potential to significantly improve the safety and user experience of software systems. SJava is the first system for checking that applications are self-stabilizing. A developer simply annotates the source code to capture the flow of values. The SJava compiler then checks that incorrect values will eventually leave the program returning the program to the exact correct state. Our experience indicates that this approach can successfully check self-stabilization of our benchmark applications.

## Acknowledgments

This research was supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350. We would like to thank Patrick Lam, Harry Xu, Philip Reames, Philippe Suter, and the anonymous reviewers for their helpful comments. We would like to thank James Jenista for help with the experiments.

## References

[1] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, volume 3385, pages 113–129. 2005.

[2] J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 101–112, 2008.

[3] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 415–426, 2006.

[4] B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–330, 2007.

[5] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, pages 176–185, 2005.

[6] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, November 1974.

[7] S. Dolev, Y. Haviv, and M. Sagiv. Self-stabilization preserving compiler. *ACM Transactions on Programming Languages and Systems*, 31:22:1–22:42, August 2009.

[8] S. Dolev and Y. A. Haviv. Self-stabilizing microprocessor: Analyzing and overcoming soft errors. *IEEE Transactions on Computers*, 55:385–399, 2006.

[9] S. Dolev and R. Yagel. Toward self-stabilizing operating systems. In *Proceedings of the 15th International Conference on Database and Expert Systems Applications*, pages 684 – 688, 2004.

[10] M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 2002.

[11] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, pages 354–378, 2010.

[12] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[13] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[14] M. C. Rinard. Living in the comfort zone. In *Proceeding of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

[15] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5 – 19, January 2003.

[16] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, 2011.

[17] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[18] F. Spoto, F. Mesnard, and E. Payet. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32:8:1–8:70, March 2010.

[19] P. Wadler. Linear types can change the world! In *Proceedings of the International Conference on Programming Concepts and Methods*. North, 1990.