Title
Self-stabilizing Java: Tool Support for Building Robust Software

Permalink
https://escholarship.org/uc/item/5k78w24c

Author
Eom, Yong hun

Publication Date
2016

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA, IRVINE

Self-stabilizing Java: Tool Support for Building Robust Software

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Yong hun Eom

Dissertation Committee:
Associate Professor Brian Demsky, Chair
Professor Pai H. Chou
Associate Professor Rainer Dömer

2016
TABLE OF CONTENTS

LIST OF FIGURES vii
ACKNOWLEDGMENTS ix
CURRICULUM VITAE x
ABSTRACT OF THE DISSERTATION xii

1 Introduction 1

1.1 Self-stabilizing Java ........................................... 2
   1.1.1 Basic Approach ........................................... 3
   1.1.2 Error Model ............................................. 4
   1.1.3 Relation to Previous Approaches ......................... 5

1.2 Usage Scenarios ............................................ 5

1.3 Contributions ............................................. 6

1.4 Organization ............................................... 7
2 Overview

2.1 Example .................................................. 11
   2.1.1 Program Summary .................................. 11
   2.1.2 Self-stabilization .................................. 11

2.2 SJava Annotations ...................................... 13
   2.2.1 Location Lattice Definition ....................... 14
   2.2.2 Type Declarations .................................. 15
   2.2.3 Composite Locations ............................... 16

2.3 Checking Self-Stabilization ........................... 16
   2.3.1 Flow-down Rule .................................... 16
   2.3.2 Value Eviction and Other Checks .................. 18

2.4 Annotation Inference .................................. 18
   2.4.1 Generating Value Flow Graph ...................... 19
   2.4.2 Generating Hierarchy Graph ....................... 19
   2.4.3 Generating Lattices ............................... 20
   2.4.4 Discussion ......................................... 21

3 Location Type System ................................. 22

3.1 Location Types ........................................ 22
3.2 Location Type Lattice .................................. 23
3.3 Method and Field Location Lattices .................. 24
3.4 Composite Location Types ............................. 24
   3.4.1 Comparison ......................................... 25
   3.4.2 Greatest Lower Bound .............................. 26
3.5 Inheritance ............................................. 27
3.6 Location Type Annotations ............................ 27
4 Value Flow-based Self-Stabilization

4.1 Flow-down Rule ...................................................... 31
  4.1.1 Notation ........................................................... 31
  4.1.2 Location Type Checking Rules ................................. 31
  4.1.3 Arrays .............................................................. 33
  4.1.4 Implicit Flow ....................................................... 34
  4.1.5 Method Invocation ............................................... 35
  4.1.6 Objects ............................................................ 38
  4.1.7 Delta Locations ................................................... 41
  4.1.8 Shared Locations ................................................ 42

4.2 Eviction of Values .................................................... 43
  4.2.1 Definitely-Written Analysis .................................... 43
  4.2.2 Shared Location Extension ..................................... 47

4.3 Termination of Event Loop Iterations ............................... 48
  4.3.1 Loop Termination Analysis ..................................... 49
  4.3.2 Loop Termination Annotations ................................. 49

4.4 Code Generation ....................................................... 50

4.5 Correctness .......................................................... 50
5 Inference Algorithm

5.1 Overview .................................................. 53
  5.1.1 Correctness Properties .......................... 53
  5.1.2 Simplification Goals .............................. 53

5.2 Annotation Inference Algorithm .................. 54
  5.2.1 Value Flow Graph .............................. 56
  5.2.2 Avoiding Unnecessary Cycles ................. 61
  5.2.3 Inferring Program Counter Locations .......... 64
  5.2.4 Return Values .................................. 65
  5.2.5 Hierarchy Graph ................................ 66
  5.2.6 Converting the Hierarchy Graphs into Lattices .. 69
  5.2.7 Non-self-stabilizing Programs ................. 69
  5.2.8 Discussion .................................. 71

5.3 Simplification ........................................ 71
  5.3.1 Constructing Interface Hierarchy Graphs ........ 73
  5.3.2 Simplifying the Interface Hierarchy Graph .... 73
  5.3.3 Inserting Merge Points ......................... 74
  5.3.4 Converting the Interface Hierarchy Graphs into Lattices 75
  5.3.5 Inserting Local Variable Locations ............ 75
6 Evaluation

6.1 Benchmarks ......................................................... 79
6.2 Self-Stabilization Evaluation ................................. 81
   6.2.1 MP3 Decoder .................................. 81
   6.2.2 Eye Tracking .................................. 83
   6.2.3 Robot Control ................................ 84
   6.2.4 Annotation Effort .............................. 85
6.3 Inference Evaluation ........................................... 86
   6.3.1 Methodology .................................. 86
   6.3.2 MP3 Decoder .................................. 87
   6.3.3 Eye Tracking .................................. 90
   6.3.4 Robot Controller .............................. 90

7 Related Work .......................................................... 91

7.1 Approaches to Robust Software Systems ......................... 91
   7.1.1 Eliminating Failures ............................ 92
   7.1.2 Failure Recovery ............................... 93
   7.1.3 Ignoring Failures ............................... 94
7.2 Self-Stabilization for Fault Tolerance ......................... 94
7.3 Language-based Information Flow and Inference ................ 95

8 Conclusion and Future Work ..................................... 97

Bibliography ........................................................... 100
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Trace of the Program State</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Manually Annotated Wind Direction Sensor Example</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Ordering Hierarchy defined for the windDirection() method</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Value Flow Graph and Hierarchy Graph for the Example</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Annotations</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Greatest Lower Bound</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Location Declaration and Annotation Grammar</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Location Type Checking Rules</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Method Invocation</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Aliasing Problem</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>Transfer Functions for Computing Read and Write Sets</td>
<td>44</td>
</tr>
<tr>
<td>4.5</td>
<td>Auxiliary Operators and Functions</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Weather Index Example</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>Transfer Functions for Computing Heap Paths</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>Transfer Functions for Flow Graph</td>
<td>58</td>
</tr>
</tbody>
</table>
5.4 Auxiliary Operators .................................................. 59
5.5 Value Flow Graph for the Example .................................. 60
5.6 Superfluous Cycle ....................................................... 62
5.7 Method Hierarchy ....................................................... 68
5.8 Field Hierarchy ......................................................... 68
5.9 Field Lattice ............................................................ 70
5.10 Simplified Field Lattice ............................................... 70
5.11 Lattice for the SynthesisFilter ....................................... 72
5.12 Merge Point ............................................................ 74
5.13 Field Interface Hierarchy .............................................. 75
5.14 Hierarchy Graph and Lattice ......................................... 77
5.15 Weather Index Example with Inferred Annotations .......... 78

6.1 Distribution of the number of output samples required for the program to return to the normal behavior after an error injection. ............................................ 82
6.2 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. ............................................ 82
6.3 Number and Type of Annotations ...................................... 84
6.4 The Lattice of Synthesis Filter ......................................... 89
ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor and committee chair Brian Demsky. I have been fortunate to have an advisor who spent countless hours discussing research ideas with me. This dissertation could not have been finished without his guidance, advice, support and encouragement.

I would also like to extend my appreciation to my committee members: Pai Chou and Rainer Dömer for their advice and support during my Ph.D. study.

I am grateful to my colleagues Alokika Dash, James Jenista, Jin Zhou, Brian Norris, Byron Hawkins, Peizhao Ou, Bin Xu, Kyoungwon Kim, and Changik Lee for their support.

This project was partly supported by the National Science Foundation under grants CCF-0846195, CCF-1217854, CNS-1228995, and CCF-1319786.

I thank ACM SIGPLAN and IEEE for permission to include content previously published.

Lastly, and above all, I owe much gratitude to my family for their endless support. I also acknowledge my friend Jungho Ham whose world-class BBQ has helped me stay sane through difficult years, and TJ for your great contribution to my well-being.
CURRICULUM VITAE

Yong hun Eom

Education

• B.E. Computer Engineering, Hongik University, Korea, 2001.

• M.S. Computer Science, Seoul National University, Korea, 2006.
  – Adviser: Suk-In Yoo.

• Ph.D. Electrical and Computer Engineering, University of California, Irvine, 2016.
  – Self-stabilizing Java: Tool Support for Building Robust Software.
  – Adviser: Brian Demsky.

Research

Research Fields

• Programming Languages

• Software Engineering

Publications

• Few-to-Many: Incremental Parallelism to Reduce Tail Latency in Interactive Services, Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, Kathryn S. McKinley, Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015.

• Sinfer: Inferring information flow lattices for checking self-stabilization, Yong hun Eom, Brian Demsky, IEEE International Symposium on Software Reliability Engineering (ISSRE), 2013

• Self-Stabilizing Java, Yong hun Eom, Brian Demsky, ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI), 2012.
• DOJ: Dynamically Parallelizing Object-Oriented Programs, Yong hun Eom, Stephen Yang, James C. Jenista, Brian Demsky, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2012.

• Using Disjoint Reachability for Parallelization, James C. Jenista, Yong hun Eom, Brian Demsky, International Conference on Compiler Construction (CC), 2011.


Peer Review

ABSTRACT OF THE DISSERTATION

Self-stabilizing Java: Tool Support for Building Robust Software

By

Yong hun Eom

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2016

Associate Professor Brian Demsky, Chair

Developing robust software systems remains an open research problem. The current approaches for improving software reliability mainly focus on minimizing the number of software bugs through formal verification or extensive testing. Despite such efforts, it is common that unexpected software bugs corrupt a program’s state and cause systems to fail.

The motivation for this research is to embrace the fact that it is difficult to guarantee that software is error-free. We present Self-stabilizing Java (SJava) that instead checks that a program self-stabilizes. Self-stabilizing programs automatically recover to the correct state from the corrupted state caused by software bugs and other sources. A number of applications are inherently self-stabilizing—such programs typically overwrite all non-constant data with new input data.

We have developed a type system and static analyses that together check whether program executions eventually transition from incorrect states to the correct state. We combine this with a code-generation strategy that ensures that a program continues executing long enough to self-stabilize. Furthermore, in order to lower the burden of type annotations, we present an annotation inference algorithm that automatically derives an initial set of annotations.
Our experience using SJava indicates that our system successfully checked that several benchmarks were self-stabilizing and effectively inferred annotations for our benchmarks.
Chapter 1

Introduction

Bugs have long plagued software systems. Researchers have primarily focused on two fundamentally different approaches for addressing this difficult problem: software verification and software testing. Software verification seeks to determine whether a software program meets a full or partial specification. Verification has seen success in determining the absence of specific types of bugs. The more ambitious goal of determining that a program faithfully implements a full specification has proven prohibitively expensive for most programs. Software verification tools from the research community typically require extensive interactions with experts. For example, an expert may have to guide a theorem prover in order to prove critical theorems. Moreover, obtaining a detailed, correct specification for a software system is itself a difficult task. As a result, full formal verification remains limited to safety critical systems that are both relatively simple and can justify the significant extra expenditures required to prove that they faithfully implement a specification.

Because of the limitations of formal verification, practitioners often rely upon testing to discover bugs in software programs. The basic idea is to use test cases to ensure that a
software program works for specific scenarios. Traditional testing, by its nature, only shows that a program works correctly in specific scenarios and execution environments. By increasing the number of scenarios explored by test cases, developers can increase confidence that a program will work for a given real-world scenario. However, traditional testing never guarantees the absence of bugs.

With testing, it is always possible that a new input will trigger a latent bug that corrupts the state of the program. After a bug corrupts the state of the program, it is possible that key invariants are violated, causing the program to behave arbitrarily. Despite extensive testing in industry, both unusual and relatively common inputs frequently trigger bugs.

The problem with previous approaches is that they cannot guarantee that software systems will resume normal behavior after software bugs or hardware failures. This dissertation presents a new approach for creating robust software systems. Our system statically checks whether programs are guaranteed to recover from an arbitrary state corruption caused by software bugs or other events and eventually reach the correct state. This property is known as self-stabilization in the field of distributed computing.

1.1 Self-stabilizing Java

We presents Self-stabilizing Java (SJava) [22], a combination of a type system and static analyses that checks whether 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.
1.1.1 Basic Approach

SJava checks that program executions eventually transition from incorrect states to the correct state by ensuring 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 the location types, which 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 (henceforth referred to as the flow-down rule). Similar to information flow [36], 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 1.1 presents a graphical depiction of the effect of these two properties on an execution whose state is corrupted by a bug. The red $\times$ symbols indicate corrupted values whereas the green $\checkmark$ symbols 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.

1.1.2 Error Model

SJava checks that applications self-stabilize in response to errors that occur inside the event loop. We assume that the application successfully reaches the entrance of the event loop. We believe that this is a reasonable assumption; a human can often intervene when systems 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 an incorrect state is fed a 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 [18, 19, 17].
1.1.3 Relation to Previous Approaches

Self-stabilization provides a middle road between testing and verification—self-stabilizing programs are guaranteed to recover from state corruption and reach the correct state in a bounded number of steps [16].

Designing programs that self-stabilizes is intended to complement, and not replace the existing approaches for software reliability. For example, we expect that software developers would still perform the same testing procedures on programs. Self-stabilization gives developers more mileage out of the same testing process; even if the testing process misses a bug, the effects of that bug are guaranteed to be limited in time. After a sufficient period of time, if the program is returned to its comfort zone [44], the program will resume its normal behavior. More precisely, after the program self-stabilizes, it will have the same behavior as that during the testing for the test inputs. Of course, if the system receives further fault-revealing inputs, the user will have to wait for it to self-stabilize again.

1.2 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.

### 1.3 Contributions

This dissertation makes the following contributions:

**New Approach for Building Robust Software:** We present a new approach for checking whether programs are self-stabilizing. Our approach guarantees that software systems will resume normal behavior after a bounded time period following an error.

**Types for Self-Stabilization:** We present a type system that ensures that values eventually flow out of a program to return the program to the correct state.
Eviction Analysis: We present a static analysis for checking the eviction of values in heap-based data structures.

Annotation Inference Algorithm: We present an annotation inference algorithm that can automatically infer a set of information flow annotations that are sufficient to check whether a program self-stabilizes. Inferring information flow annotations is also potentially useful for checking security properties.

An Approach for Simplifying Annotations: We present a simplification technique that reduces the complexity of the lattices. It improves the understandability of the lattices and makes our technique useful for other domains such as program understanding and debugging.

Implementation and Experience: We have implemented an SJava compiler including the type system, the static analyses, and the annotation inference algorithm and reported our experience with three benchmark applications.

1.4 Organization

This dissertation is organized as follows.

Chapter 2 presents an example program with manual annotations and the process by SJava checker. Chapter 4 presents the steps of the flow-based self-stabilization check in details. Chapter 5 presents the annotation inference algorithm. Chapter 6 evaluates SJava on three existing application. Chapter 7 discusses related work. We discuss the future work and conclude in Chapter 8.
Chapter 2

Overview

Our basic approach for checking self-stabilization is to statically reason about value flows throughout a program to check that incorrect values eventually leave and are replaced by correct values. We first map a program’s memory locations into abstract locations using location types. The location type system then establishes an ordering relation on abstract memory locations in the program, which constrains how values flow through the program. For capturing value flows in the program, SJava requires developers to annotate code using location types from either scratch or the initial set of annotations that is automatically generated by the inference algorithm.

SJava checks the following two properties. First, values only flow from a higher to a lower abstract location, which is referred as the flow-down rule in Section 1.1.1. Second, values can only remain at a given location for a bounded time. Two properties together guarantee that after a period of time, the highest levels of the lattice must have correct values because they must be overwritten and can only depend on new input data. As the execution progresses, memory locations with increasingly lower location types have the corrupted value overwritten with the correct value because values only flow down.
Ultimately, all memory locations have the correct value and the execution self-stabilizes into the correct state.

One potential problem of the location type system is that unrestricted aliasing could be used to circumvent the location type system. SJava uses a linear type system [52] to ensure that a given object cannot be aliased by two different references with different heap locations.

Having discussed how to check that the program will return to the correct state after a bounded number of main event loop iteration, we also need to show that each iteration of main event loop terminates. If the main event loop does not terminate, the program never self-stabilizes. We present the simple termination analysis that checks common terminating loop patterns.

SJava guarantees that the program will return to normal behavior if an execution continues for a sufficiently long period of time. However, if the program crashes upon failure, it never self-stabilizes. SJava provides an option that ignores error. SJava compiler infrastructure will assign a legal behavior to an operation that raises an uncaught exception (i.e., reading a reference field of a null reference will simply return null). Since the program checked by SJava flushes out all non-loop-invariant values within a bounded time, the effect caused by ignoring the error is limited.

We summarize our approach into the list of key components as follows.

- **Arrange State into a Lattice:** The approach uses annotations to define a location lattice. The location lattice contains edges that constrain how information flows between memory locations in the program. The location lattice is acyclic, prohibiting cyclic information flows. In SJava, every memory location (fields, arrays and
variables) has a location type in addition to its Java type. The programmer uses location type annotations to map memory locations to lattice locations.

- **Constrain Information to only Flow Down this Lattice:** A type checker then checks that assignments only flow information down this lattice, which is referred to as the *flow-down rule*. The approach enforces the flow constraint on both the explicit flows caused by assignments and the implicit flows caused by conditional branches.

- **Eviction:** 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 the main event loop. 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.

- **Termination:** A termination analysis ensures that each iteration of the event loop terminates by prohibiting recursive calls \(^1\) and checking that inner loops terminate.

- **Crash Avoidance:** 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.

- **Annotation Inference:** SJava requires developers to manually annotate code with location types. To lower the annotation burden of developers, SJava provides an annotation inference algorithm that automatically generates an initial set of

\(^1\)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.
annotations. It eliminates a key manual step in the process by automatically inferring sufficient annotations to check self-stabilization.

Next we present an example code to which we will apply SJava to check self-stabilization. Afterwards, we explain each component of the system in detail along with an example.

2.1 Example

This section presents an example program and why this program self-stabilizes.

2.1.1 Program Summary

We will use a simple wind direction sensor example to illustrate the approach. Figure 2.1 presents the main event loop for the example. The main event loop reads the current wind direction in Line 11, 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.

2.1.2 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 sensor device returns an erroneous value, which is not one of the 16 possible directions. This value is stored in and corrupts the `bin` data structure in Line 16. Next, the `convertToString()` method in Line 31 throws an
@LATTICE("DIR<TMP,TMP<BIN")
public class WDSensor{
    @LOC("BIN") private WindRec bin = new WindRec();
    @LOC("DIR") private int dir;
    @LATTICE("STR<WDOBJ,WDOBJ<IN")
    @THISLOC("WDOBJ")
    public void windDirection(){
        SJAVA:
            while(true){ // main event loop
                @LOC("IN") int inDir = Device.readSensor();
                // move old wind directions one step down
                bin.dir2 = windRec.dir1;
                bin.dir1 = windRec.dir0;
                // add a new wind direction
                bin.dir0 = inDir;
                @LOC("STR") String outDir = calculate();
                broadcastChange(outDir);
            }
    }
    @LATTICE("OUT<CAOBJ")
    @THISLOC("CAOBJ")
    @RETURNLOC("OUT")
    public String calculate(){
        @LOC("CAOBJ,TMP") int majorDir;
        @LOC("OUT") String strDir;
        // calculate the majority
        ...
        this.dir = majorDir;
        strDir = convertToString(majorDir);
        return strDir;
    }
}
@LATTICE("DIR2<DIR1,DIR1<DIR0")
class WindRec{
    @LOC("DIR0") public int dir0;
    @LOC("DIR1") public int dir1;
    @LOC("DIR2") public int dir2;
}

Figure 2.1: Manually Annotated Wind Direction Sensor Example
uncaught null pointer exception due to the corrupted value in the bin data structure. 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. However, 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 bin data structure, the program would return to the correct execution after, at most, three iterations of the main loop. This happens as long as the main event loop retrieves a new valid direction in Line 11 because the program eventually overwrites incorrect values with new correct values in all memory locations.

2.2 SJava Annotations

To statically check if a program self-stabilizes, SJava requires developers to manually annotate the program’s code with location types. The purpose of annotations is to capture the flow of values through the program. The SJava compiler then checks that incorrect values will eventually leave the program, returning the program to the exact correct state. Figure 2.1 presents the example with manual annotations.

The SJava location type system performs three functions. First, it provides a static location lattice that will be used to statically partition the program’s state. Second, a set of type declaration annotations maps the program’s state onto locations in the location lattice. Together, these components arrange the program’s state and encode how information is allowed to flow between memory locations. Finally, the type rules check that all information flows in the program respect the location lattice. We next describe each of these components in more detail.
2.2.1 Location Lattice Definition

The location lattice defines the set of location types and orders them. SJJava has two types of location lattices: method lattices and field lattices. Method lattices define a method local ordering of the methods’ parameters and local variables. Field lattices define an ordering of the classes’ fields.

Method lattices are declared with the annotation @LATTICE\(^2\), which appears immediately before the corresponding method declaration. This annotation uses the lower than (<) operator to order two location types. Location types are implicitly declared by using them in an expression with the lower than operator in the lattice declaration. Line 6 of Figure 2.1 presents the method lattice for the windDirection() method, which represents an ordering hierarchy as shown in Figure 2.2. For example, the @LATTICE annotation declares two method location types, WDOBJ and STR, and orders the type WDOBJ higher than the type STR. This declaration means that information can only flow from locations with the type WDOBJ to locations with the type STR.

\[ \text{IN} \]
\[ \text{WDOBJ} \]
\[ \text{STR} \]

**Figure 2.2:** Ordering Hierarchy defined for the windDirection() method

---

\(^2\)SJJava does not add any new syntax to Java. Instead, SJJava uses the existing Java annotation framework and loop labels for its annotations. As a result, SJJava programs are legal Java programs.
Each method has its own method lattice. This provides the necessary compositionality for libraries and other software components to be developed and annotated independently from the code that uses them. Compatibility of the method lattices between a caller and a callee is then checked at the call sites.

Simply using the method lattice would result in all fields of a given object having the location of the object’s reference. This would prevent any flow of information between different fields of the same object. To eliminate this limitation, SJava includes field lattices that order different fields of the same object. The field location lattice is declared using the same basic syntax as a method lattice, but the field lattice declaration appears immediately before a class declaration. Line 1 of Figure 2.1 presents the field lattice declaration for the WDSensor class.

### 2.2.2 Type Declarations

SJJava requires variable, parameter, and field declarations to have location types. The annotation @LOC is used to declare the location type of a parameter, field, or variable. The annotation @LOC("IN") in Line 11 of Figure 2.1 declares that the variable inDir has the location type IN. The location type of a parameter is declared using the same syntax immediately before the type declaration of the parameter. The location type of the implicit this parameter is declared using the @THISLOC annotation. For example, the annotation @THISLOC("WDOBJ") in Line 7 declares that the this variable has the location type WDOBJ.

The location types of fields are declared using the same syntax as variables. The annotation @LOC("BIN") in Line 3 of Figure 2.1 declares that the field bin has the location type BIN.


2.2.3 Composite Locations

Method and field lattices are combined into a composite lattice with the ordering of the composite lattice determined by lexical ordering of the component lattices. Each type in a composite lattice begins with an element from the current method’s location lattice, followed by elements from field location lattices.

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 \( \text{LOC} \langle \text{CAOBJ}, \text{TMP} \rangle \) for the \texttt{majorDir} variable. This location is lower than \( \langle \text{CAOBJ}, \text{BIN} \rangle \) and higher than \( \langle \text{CAOBJ}, \text{DIR} \rangle \) since the field hierarchy has the ordering relations \( \text{DIR} \sqsubseteq \text{TMP} \) and \( \text{TMP} \sqsubseteq \text{BIN} \).

Lastly, the special label \texttt{SJJAVA} in Line 9 specifies that the \texttt{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

This section describes how SJJava uses the collection of type checks and static analyses to check if the example code self-stabilizes.

2.3.1 Flow-down Rule

The SJJava 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.dir`
in line 30 is valid because the location type `(CAOBJ,TMP)` of the source value is higher than
the location type `(CAOBJ,DIR)` of the destination. Note that the comparison of two
locations is based on lexicographical ordering of the location elements.

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 in
Figure 2.1 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 `outDir` has a
lower location than the `this` variable).
2.3.2 Value Eviction and Other Checks

The flow-down rule by itself does not ensure self-stabilization. This rule may allow corrupt values to remain in a location indefinitely. To address this issue, SJava uses an eviction check, which ensures that locations are either (1) loop invariant, (2) overwritten in every loop iteration, or (3) overwritten before they are read. Satisfying any of these three properties ensures that the program cannot read stale, corrupt values. SJava uses static analysis to check these properties. In this example, the main event loop overwrites all non-loop-invariant memory locations, such as the fields of the class WindRec (dir0, dir1, dir2) at every iteration. Section 4.2 will present the eviction analysis in detail.

Unrestricted aliasing could be used to circumvent the type system. SJava uses linear types\[52\] to ensure that a given object cannot be aliased by two different references with different heap locations. Section 4.1.6 will present our linear type system in detail.

2.4 Annotation Inference

In this section, we briefly present how the annotation inference algorithm automatically derives an initial set of annotations for the example in Figure 2.1. Although the example does not rely on a complicated set of annotations, the annotation still requires additional efforts of developments. Moreover, when checking legacy system, developers often face difficulties to annotate codes to check self-stabilization because they first need to understand how information flows through the program. Understanding the flow of information in legacy systems is a time consuming job and might present a significant barrier to adoption.
2.4.1 Generating Value Flow Graph

The annotation inference algorithm begins by generating a graph that models how values flow within each method, which is called a *value flow graph*. To illustrate the inference algorithm, we will use the example from Figure 2.1 with annotations removed. Figure 2.3 presents the value flow graph for the example. A node in a value flow graph represents initial approximation of location type assignment. An edge represents value flows.

The value flow graph construction algorithm initially assumes that all variables and parameters will have location types that contain a method variable type and no field location types. As the annotation generation algorithm discovers problems with these initial location type assignments, it will refine the location type of variables to include field location type members. Section 5.2.1 presents more details for generating value flow graphs.

2.4.2 Generating Hierarchy Graph

At this point, information flows in the value flow graph are not modeled in a modular manner — edges correspond to flows in both field and method lattices. However, as described in Section 2.2.1, the location type system defines individual lattice for each method and class. Therefore, the algorithm decomposes value flows in the value flow graph into method and field flows, which are abstracted by a method and field hierarchy graph, respectively.

The right side of Figure 2.3 shows the method hierarchy graph for the `windDirection()` method and the field hierarchy graph for the `WindRec` class. We will present the details of the hierarchy graph construction in Section 5.2.5.
2.4.3 Generating Lattices

In this example, two hierarchy graphs in Figure 2.3 capture the information flows between all memory locations in the program. While the hierarchy graphs are partial orders, they may not be lattices, as the GLB and LUB are not necessarily well defined. Therefore, we
convert them into lattices by adding extra nodes to make the GLB and LUB well defined.

The problem of finding the smallest complete lattice that contains a partial order is known as the Dedekind-MacNeille completion. We use the algorithm developed by Nourine and Raynaud [38] to compute the Dedekind-MacNeille completion of the field and method hierarchy graphs to generate field and method lattices, respectively.

Since two hierarchy graphs in this example represent relatively simple partial orders, Section 5.2.6 will present an interesting example for generating lattices.

### 2.4.4 Discussion

Our approach for inferring annotations in this example simply captures the flow of values in the example code and then generates lattices that precisely represent the flow of values between each memory location (i.e., fields and variables) in the program. The problem is that this approach is the only feasible solution for a very simple program because, in practice, a program often defines a large number of memory locations and information paths. Chapter 5 will present our techniques for dealing with a more complicated program code.
Chapter 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, \( \text{low} \sqsubset \text{high} \) means that the location \text{high} is higher than \text{low}, specifying that values can legally flow from memory locations with the location type \text{high} to memory locations with the location type \text{low}. The location lattice includes the top and bottom locations. The top location \( \top \) is the highest location, whose values can flow anywhere. The bottom location \( \bot \) 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 ⟨FOO, Foo.BAR, Bar.Z⟩, 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 $\{\sqsubseteq_1, \sqsubseteq_2, \ldots, \sqsubseteq_n\}$. The partial ordering relation of the composite location is defined as follows:

\[
\langle a_1, a_2, \ldots, a_n \rangle \sqsubseteq_C \langle b_1, b_2, \ldots, b_n \rangle \iff \\
\exists j \in \{1, \ldots, n\}, (a_j \sqsubseteq_j b_j \lor (j = n \land a_j = b_j)) \land \forall i < j. a_i = b_i
\] (3.1)

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.
\[ \text{GLB}(\langle a_1, a_2, ..., a_n \rangle, \langle b_1, b_2, ..., b_n \rangle) \]

1. \( g_1 = \text{glb}(a_1, b_1) \)
2. \( \text{if } g_1 \neq a_1 \land g_1 \neq b_1 \text{ then } \ // \text{case 1} \)
3. \( \text{for } i = 2 \text{ to } n \)
4. \( g_i = \bot \)
5. \( \text{else if } g_1 = a_1 \land g_1 \neq b_1 \text{ then } \ // \text{case 2} \)
6. \( \text{for } i = 2 \text{ to } n \)
7. \( g_i = a_i \)
8. \( \text{else if } g_1 \neq a_1 \land g_1 = b_1 \text{ then } \ // \text{case 3} \)
9. \( \text{for } i = 2 \text{ to } n \)
10. \( g_i = b_i \)
11. \( \text{else } \ // \text{case 4} \)
12. \( \langle g_2, g_3, ..., g_n \rangle = \text{GLB}(\langle a_2, a_3, ..., a_n \rangle, \langle b_2, b_3, ..., b_n \rangle) \)
13. \( \text{return } \langle g_1, g_2, ..., g_n \rangle \)

**Figure 3.2: Greatest Lower Bound**

### 3.4.2 Greatest Lower Bound

Calculating the greatest lower bound of two composite locations requires a meet operator over the composite lattice, where more than one method or field lattice participates in the calculation.

Figure 3.2 presents pseudo-code for a GLB function that implements the \( \cap \) operator for the composite location. Line 1 calculates the GLB of the two location elements in the first positions \( a_1 \) and \( b_1 \) using the \( \cap_1 \) operator for the lattice of those two elements. If the two elements are field elements from different classes, the GLB function returns the bottom value. If the GLB of the first element \( g_1 \) is not equal to either \( a_1 \) or \( a_2 \) (case 1) its level of the hierarchy is already lower than \( a_1 \) and \( b_1 \) so Line 4 assigns the \( \top \) location to the remaining elements. If \( g_1 \) is equal to either of the first elements \( a_1 \) or \( b_1 \) (case 2 and case 3) the GLB is set to the input composite location satisfying this condition. The last case (case 4) occurs when \( g_1 \) is equal to both \( a_1 \) and \( b_1 \). In this case, the procedure recursively calls itself to compute the GLB of the remaining elements.
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 3.1 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 3.3 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 $x$.

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.
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 3.3: Location Declaration and Annotation Grammar

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, from Figure 2.1 in Section 2.1.1, developers can declare a new variable `newDir` with the annotation `@LOC("WDOBJ,DIR0") int newDir`. It indicates that the variable `newDir` has the composite location type ⟨WDOBJ,DIR0⟩. 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.
Chapter 4

Value Flow-based Self-Stabilization

This chapter presents the value flow-based self-stabilization that checks if program executions eventually transition from incorrect states to the correct states. Our system consists of a type system and static analyses. We use a type system to arrange the programs state into a lattice of locations and then show that information can only flow down this lattice. Static analyses then ensures that the program evicts values from a given location within a bounded time.

Section 4.1 describes type checking rules. Section 4.2 describes static analyses for the eviction checking. Section 4.3 describes how SJava ensures that each iteration of the main event loop terminates. Section 4.4 describes our compiler infrastructure for crash avoidance. Section 4.5 presents the correctness argument for the value flow-based self-stabilization.
4.1 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.

4.1.1 Notation

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 $(l_0, l_1, ..., l_{n-1})$. 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.

4.1.2 Location Type Checking Rules

Figure 4.1 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:
Figure 4.1: Location Type Checking Rules
**Literal:** Every literal value has the highest location type \( \text{TOP} \) in the location hierarchy, denoted by \( T \). 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 \( e_0 \boxplus e_1 \). 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 \texttt{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.

4.1.3 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 4.1.8 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.

4.1.4 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.

\[
\text{if (a>0) } b=1; \text{ else } b=0;
\]
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 ⟨A⟩, 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.

### 4.1.5 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.
The parameter’s location type describes how the location type of an argument in the caller is transferred into the method hierarchy 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 $p_i \sqsubseteq p_j$ between parameters, then the caller has to have the corresponding ordering relation $a_i \sqsubseteq a_j$ 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 4.2 illustrates how the compiler checks constraints of the caller and the callee. The callee has two ordering relations among parameters, ENV ⊑ IN and DAOBJ ⊑ 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 ENV ⊑ VA and DATA ⊑ 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 ⟨IOBJ, F⟩ and the receiver object has the location type IOBJ. The corresponding argument in the caller is required to be higher than or equal to ⟨O, F⟩ if the object whose method is being invoked has the location type ⟨O⟩ in the caller.

**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 4.1 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 location 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 4.2, 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 4.2 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 4.2 satisfies the ordering constraints with the location type of the return value DATA.

4.1.6 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.
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. Linear types have been leveraged in programming languages[52, 25, 27] for various purposes including safe concurrent programming, resource management, and protocol checking. 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.

**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.

### 4.1.7 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 \textit{delta} that takes a composite
location and generates a new composite location, called a \textit{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 2.1, the composite location
\langle WDOBJ, DIR1 \rangle can be replaced with \textit{delta}(\langle WDOBJ, DIO0 \rangle). It generates a new location that
is lower than \langle WDOBJ, DIO0 \rangle and higher than all locations below \langle WDOBJ, DIR1 \rangle.

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.
4.1.8 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.
4.2 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.

4.2.1 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.

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.
\[
\begin{align*}
\text{st} & \quad x = y \cdot f \\
H P' &= H P \cup \{ (x, p \oplus f) \mid p \in H P(y) \} \\
R_{\text{new}} &= \{ p \oplus f \mid p \in H P(y), \exists p' \in WT \Rightarrow \neg \text{Pre}(p \oplus f, p') \} \\
R_m' &= R_m \cup R_{\text{new}} \\
\text{x.f=y} & \\
W T' &= W T \cup \{ H P(x) \oplus f \} \\
O W_{m'} &= O W_m \cup \{ H P(x) \oplus f \} \\
\text{call} & \quad c(a_0, \ldots, a_n) \\
R_{\text{bound}}^c &= \bigcup_{c \in \text{calleeSet}(c), i \in \{0, \ldots, n\}} \{ H P(a_i) \odot r \mid r \in R^c \land \text{Eq}(r, p_i) \} \\
O W_{\text{bound}}^c &= \bigcup_{c \in \text{calleeSet}(c), i \in \{0, \ldots, n\}} \{ H P(a_i) \odot r \mid r \in O W^c \land \text{Eq}(r, p_i) \} \\
W T_{\text{bound}}^c &= \bigcap_{c \in \text{calleeSet}(c), i \in \{0, \ldots, n\}} \{ H P(a_i) \odot r \mid r \in W T^c \land \text{Eq}(r, p_i) \} \\
R_{\text{new}} &= \{ p \mid p \in R_{\text{bound}}^c, \exists p' \in W T \Rightarrow \neg \text{Pre}(p, p') \} \\
R_m' &= R_m \cup R_{\text{new}} \\
O W_{m'} &= O W_m \cup O W_{\text{bound}}^c \\
W T' &= W T \cup W T_{\text{bound}}^c \\
\text{merge} & \\
W T' &= \bigcap_{i \in \text{pred}(st)} W T_i \\
\text{exit} & \\
W T_m &= \bigcap_{i \in \text{pred}(st)} W T_i
\end{align*}
\]

\textbf{Figure 4.4:} Transfer Functions for Computing Read and Write Sets
\langle a_0, a_1, \ldots, a_n \rangle \oplus b = \langle a_0, \ldots, a_n, b \rangle \\
\langle a_0, a_1, \ldots, a_n \rangle \odot \langle b_0, b_1, \ldots, b_n \rangle = \langle a_0, \ldots, a_n, b_1, \ldots, b_n \rangle \\
Eq(\langle a_0, \ldots, a_n \rangle, \langle b_0, \ldots, b_n \rangle) = (a_0 = b_0) \\
Pre(\langle a_0, \ldots, a_n \rangle, \langle b_0, \ldots, b_k \rangle) = k \leq n \land (\langle a_0, \ldots, a_k \rangle = \langle b_0, \ldots, b_k \rangle)

Figure 4.5: Auxiliary Operators and Functions

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 \( p_1 \) through a sequence of references \( r_1, \ldots, r_n \) generates the
heap path \( \langle p_1, r_1, \ldots, r_n, f \rangle \). 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 4.4 presents the transfer
functions for computing read and write set. We define the helper function
\[ HP(x) = \{ hp | \langle x, hp \rangle \in HP \} \].

**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 \( HP(y) \). 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 \( c(a_0, ..., a_n) \), the callee’s read and write effects are propagated to the caller. The analysis first computes the sets \( OW^c_{\text{bound}} \), \( WT^c_{\text{bound}} \) and \( R^c_{\text{bound}} \) for all possible callees. 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 \( \langle d, g \rangle \) that is passed as the parameter \( x \) to the callee \( c \). If the callee has the two read tuples \( \langle x, y, a \rangle \) and \( \langle x, y, b \rangle \) in its set \( R^c \), the corresponding caller context read tuples \( \langle d, g, y, a \rangle \) and \( \langle d, g, y, b \rangle \) are added to the set \( R^c_{\text{bound}} \). The set \( R^c_{\text{bound}} \) and \( OW^c_{\text{bound}} \) are the union of all possible callees, and the set \( WT^c_{\text{bound}} \) is the intersection of all possible callees. Then, the set \( R^m \) of the caller gains an element of \( R^c_{\text{bound}} \) 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 \( OW^c_{\text{bound}} \) and \( WT^c_{\text{bound}} \), 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.

**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 \( R_{\text{new}} \) 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., \( \exists p \in WT_{st}, \text{Pre}(p, p') \), or
3. that the heap path \( p \) or some prefix is overwritten in every loop iteration, i.e., at every loop backedge statement \( st' \) \( \exists p \in WT_{st'}, \text{Pre}(p, p') \).

### 4.2.2 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 $WT$.

### 4.3 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.

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 [13, 14, 49, 9, 8]. If necessary, we could easily replace our termination analysis with more sophisticated approaches.
4.3.1 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.

4.3.2 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.

### 4.4 Code Generation

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.

### 4.5 Correctness

In this section, we sketch the basic correctness argument for SJava.
Lemma 4.5.1 (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 4.5.2 (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 4.5.3 (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 4.5.1 and 4.5.2 all non-constant memory locations will eventually have the correct values.
Chapter 5

Inference Algorithm

This chapter presents the annotation inference algorithm in SJava that automatically generates the set of initial annotations [23]. Our goal is to generate simple and human-understandable location type structure. A key advantage of a simpler location type structure is that the developer can use the generated annotations to gain a basic intuition about how information flows through the program. With a simple structure, developers can easily figure out what location type constraints the code must satisfy to self-stabilize. Simplifying lattices is therefore useful for guiding the developer in modifying the code in a way that maintains the self-stabilization property. In addition, if the developer chooses to customize the generated annotations, the simpler lattices will be easier to understand and modify.

We first identify correctness properties and simplification goals for the inference algorithm in Section 5.1. Section 5.2 presents the basic annotation inference algorithm that guarantees the correctness properties. Section 5.3 describes how to optimize inference in regarding to our simplification goals.
5.1 Overview

Our system automatically infer annotations for checking self-stabilization. We discuss both correctness properties and simplification goals for the inferred annotations.

5.1.1 Correctness Properties

We begin by stating the three correctness properties for SJava annotations:

- **Lattice Structure**: The structure of the location types forms a lattice.

- **Completeness**: Every variable, field, method parameter, method return value, or method program counter must be assigned some type, whether implicitly or explicitly.

- **Flow Constraints**: All information flows are captured by the ordering relation constraints defined by lattices.

5.1.2 Simplification Goals

Our initial implementation attempted to maintain maximally precise flow information that satisfied the correctness properties, but we found that the resulting lattices were extremely complex. This made the annotations incomprehensible to developers, and maintaining them during code revisions would not be feasible.

To reduce the complexity, we generally favor a lattice that defines the minimum number of location types necessary to maintain the correctness properties. However, this can unnecessarily constrain the external interfaces of classes and methods (e.g., parameters, return values, program counters and fields). The problem is that if the class or method were used in a different environment, developers may need to assign new location types.
Therefore, for interface members, the location types should most accurately model the value flows in the original program. Thus, the inference algorithm has the following two simplification goals:

- **Precise Interfaces**: The location types for interface members must precisely model the value flows in the program.

- **Simplicity**: For all other locations, the lattice should be as simple as possible.

A key advantage of a simpler location type structure is that the developer can use the generated annotations to gain a basic intuition about how information flows through the program. With a simple structure, developers can easily figure out what location type constraints the code must satisfy to self-stabilize. Simplifying lattices is therefore useful for guiding the developer in modify the code in a way that maintains the self-stabilization property. In addition, if the developer chooses to customize the generated annotations, the simpler lattices will be easier to understand and modify.

We will use a weather index calculation example to illustrate the approach. Figure 5.1 presents the main event loop for the example. The main event loop reads the current temperature and humidity in Line 14 and Line 15, computes the average temperature with the previous temperature in Line 17. Then, computes a weather index that combines temperature and humidity to determine the human-perceived temperature.

### 5.2 Annotation Inference Algorithm

In the previous section, we described three annotation correctness properties. To satisfy these properties, our basic approach attempts to maintain the most precise flow information. We first generate a set of constraints in the form of a directed graph where
public class Weather {
    public float prevTemp;
    public float avgTemp;
    public float curHum;
    public float index;

    // define constants c1 to c9
    public static final float c1 = -0.22475541;
...

    public void calculateIndex(){
        SSJAVA:
        while (true) {  // main event loop
    float inTemp = Device.readTemp();
    curHum = Device.readHumidity();
    // calculate the average temperature
    avgTemp = (prevTemp+inTemp)/2;
    prevTemp = inTemp;

    float f1=c1*avgTemp*curHum;
    float f2=c2*avgTemp*avgTemp;
    float f3=c3*curHum*curHum;
    float f4=c4*f2*curHum;
    float f5=c5*f3*avgTemp;
    float f6=c6*f1*f2;

    index = c7+c8*avgTemp+c9*curHum +
            f1+f2+f3+f4+f5+f6;

    broadcast(index);
    }
}

Figure 5.1: Weather Index Example
nodes are memory locations and edges represent explicit and implicit information flows. Similar to the constraint-based type inference [41], the graph captures constraints whose solution are the type annotations. Then, we perform inference and find mappings from locations to location types in the lattices that satisfy the set of constraints.

5.2.1 Value Flow Graph

The annotation inference algorithm begins by generating a value flow graph from the program source code. A node \( n \in N \) in a value flow graph represents initial approximation of a location type assignment and is represented by a tuple \( t = \langle v, f_1, ..., f_n \rangle \in T \) in which the first element of the tuple is a member of the method’s variable lattice \( v \in V \) and the subsequent elements are members of field lattices \( f \in F \). An edge \( e \in E_f \) represents value flows.

Definition 1. *(Value Flow Graph)* A method’s value flow graph is a directed graph \( G = (N, E_f) \), where a node corresponds to a location and an edge \((n_1 \rightarrow n_2) \in E_f\) corresponds to an explicit or implicit information flow from \( n_1 \) to \( n_2 \).

The value flow graph represents the following flow constraints that our system uses for type inference.

*Flow Constraint: If there exists an explicit or implicit information flow from location \( n_1 \) to location \( n_2 \), then there must exist an edge \((n_1 \rightarrow n_2)\) that indicates the location type of \( n_1 \) must be higher than the location type of \( n_2 \).*

We formulate the value flow graph construction algorithm as a dataflow analysis on the control flow graph. The construction of a control flow graph may introduce a number of temporary variables that do not require location type annotations. Our inference tool is designed to avoid transformations to intermediate representations that change which
variables are used in operations. We use the predicate `isLoc : V → boolean` to track which variables are either parameters or programmer declared and thus require annotations. The `isLoc` predicate returns true for such variables and false for all compiler introduced temporaries.

The first analysis computes a mapping \( R \subseteq V \times T \) from variables to sets of composite location tuples. This analysis also models implicit flows by using a stack \( S \) of sets of tuples \( r \in \mathcal{P}(T) \). At each conditional branch, it pushes a set of tuples onto the stack \( S \) and at each merge it pops the top set of tuples off of the stack. Figure 5.2 presents the dataflow equations for this analysis. Figure 5.4 defines auxiliary operators. For each developer-declared variable and parameter, this analysis maps the variable to a tuple that

---

<table>
<thead>
<tr>
<th>( \text{st} )</th>
<th>( R' := (R - \text{KILL}) \cup \text{GEN} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \ldots ) where ( \text{isLoc}(x) )</td>
<td>( \text{GEN} := {\langle x, \langle x \rangle \rangle} )</td>
</tr>
<tr>
<td>( x = y \cdot f ) where ( \neg \text{isLoc}(x) )</td>
<td>( \text{GEN} := {\langle x, p \oplus f \rangle</td>
</tr>
<tr>
<td>( x = y[i] ) where ( \neg \text{isLoc}(x) )</td>
<td>( \text{GEN} := {\langle x, p \oplus \text{element} \rangle</td>
</tr>
<tr>
<td>( x = y ) where ( \neg \text{isLoc}(x) )</td>
<td>( \text{KILL} := {\langle x, p \rangle</td>
</tr>
<tr>
<td>( x = \text{op}(y, z) ) where ( \neg \text{isLoc}(x) )</td>
<td>( \text{KILL} := {\langle x, p \rangle</td>
</tr>
<tr>
<td>\text{cond-branch}(x)</td>
<td>( S' := \langle R(x), S \rangle )</td>
</tr>
<tr>
<td>\text{cond-merge}</td>
<td>( S' := S_0 ) where ( S = \langle r, S_0 \rangle )</td>
</tr>
</tbody>
</table>

**Figure 5.2:** *Transfer Functions for Computing Heap Paths*
\[ E' := E \cup \text{GEN} \]

<table>
<thead>
<tr>
<th>st</th>
<th>GEN :=</th>
<th>( { \langle \langle x, p \oplus f \rangle \mid \forall p \in \mathcal{R}(y) } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y. f ) \ where ( \text{isLoc}(x) )</td>
<td>( \cup { \langle x, p \rangle \mid p \in \mathcal{S}(S) } )</td>
<td></td>
</tr>
<tr>
<td>( x = y[i] ) \ where ( \text{isLoc}(x) )</td>
<td>( \cup { \langle x, p \rangle \mid p \in \mathcal{R}(i) } )</td>
<td></td>
</tr>
<tr>
<td>( x = y ) \ where ( \text{isLoc}(x) )</td>
<td>( \cup { \langle x, p \rangle \mid p \in \mathcal{S}(S) } )</td>
<td></td>
</tr>
<tr>
<td>( x = \text{op}(y, z) ) \ where ( \text{isLoc}(x) )</td>
<td>( { \langle x, p \rangle \mid p \in \mathcal{R}(y) \cup \mathcal{R}(z) } )</td>
<td></td>
</tr>
<tr>
<td>( x.f = y )</td>
<td>( { \langle p \oplus f, p' \rangle \mid p \in \mathcal{R}(x), ) ( p' \in \mathcal{R}(y) \cup \mathcal{S}(S) } )</td>
<td></td>
</tr>
<tr>
<td>( x[i] = y )</td>
<td>( { \langle p \oplus \text{element}, p' \rangle \mid p \in \mathcal{R}(x), ) ( p' \in \mathcal{R}(y) \cup \mathcal{R}(i) \cup \mathcal{S}(S) } )</td>
<td></td>
</tr>
<tr>
<td>( x = m(y_0, y_1, \ldots, y_n) )</td>
<td>( { \langle a, a' \rangle \mid \forall i, j, 0 \leq i, j \leq n, \langle P_i, P_j \rangle \in G_m, ) ( a \in \mathcal{R}(y_i), a' \in \mathcal{R}(y_j) \cup \mathcal{S}(S) } \cup { \langle x, p \rangle \mid ) ( \forall i, 0 \leq i \leq n, \langle R, P_i \rangle \in G_m, p \in \mathcal{R}(i) \cup \mathcal{S}(S) } ) ( \cup { \langle a, a' \rangle \mid \forall (f, f') \in R^+(G_m), (P_f(f) \in P(m) ) ( \land P_f(f') \in P(m)), a \in B(f), a' \in B(f') \cup \mathcal{S}(S) } ) ( B(f) = { q \circ S_f(f) \mid \forall k, 0 \leq k \leq n, P_f(f) = P_k, q \in \mathcal{R}(y_k) } )</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.3:** Transfer Functions for Flow Graph
contains just the variable. For each compiler-introduced temporary, the analysis tracks the information used to generate its value.

The second analysis adds edges to the value flow graph to model the program’s information flows. It uses the mappings built by the first dataflow analysis. Figure 5.3 presents the dataflow equations. For every flow to a developer declared variable or an object field, the analysis generates an edge to model the flow.

The interprocedural analysis is structured as a fixed point computation. Whenever a method flow graph changes, all methods that call that method are scheduled for reanalysis. Currently, SJava constrains the program to not contain recursive call chains\(^1\). Therefore, our current implementation topologically sorts the methods and then analyzes them in topologically sorted order to avoid reanalysis of methods. But the basic analysis design supports recursive methods.

When an expression has more than one operand, the SJava type checker derives the location type of the expression by calculating the greatest lower bound (GLB) of the location types of its operands. Therefore, the corresponding lattice needs to include a location type that allows the type checker to store the GLB result. In the case of an expression that appears on the right hand side of an assignment or as a method parameter, our inference tool generates an extra node in the value flow graph, called an *intermediate*

\(\langle a_0, a_1, ..., a_n \rangle \odot \langle b_0, b_1, ..., b_n \rangle = \langle a_0, ..., a_n, b_1, ..., b_n \rangle\)

\(\langle a_0, a_1, ..., a_n \rangle \oplus b = \langle a_0, ..., a_n, b \rangle\)

\(S(\langle r_0, \langle r_1, \langle ..., r_n \rangle \rangle \rangle) = r_0 \cup r_1 \cup ... \cup r_n\)

\(Pf(\langle a_0, ..., a_n \rangle) = \langle a_0 \rangle\)

\(Sf(\langle a_0, ..., a_n \rangle) = \langle a_1, ..., a_n \rangle\)

---

\(^1\)The restriction is only necessary because the termination analysis in SJava cannot currently check the termination of recursive calls.
node, which has incoming edges from the operand nodes and an outgoing edge to the original destination of the expression.

Figure 5.5 presents the value flow graph for the example. The nodes in the graph represent locations and the edges represent flows. For example, the edge from the node labeled <inTemp> to the node labeled <this, prevTemp> indicates that information may flow from the variable inTemp to a field with the prevTemp location of an object with the this location. We group nodes with the same composite location prefix together in a rectangle. Lastly, intermediate nodes are expressed by ILOC in Figure 5.5.
5.2.2 Avoiding Unnecessary Cycles

As our algorithm proceeds, it decomposes the value flow graph into field and method hierarchies. In some cases, this decomposition can merge two different nodes in the value flow graph into the same node in a field or method hierarchy. If one of these nodes is reachable from the other, this merging has the potential to create a superfluous cycle in the lattice. These cycles can occur when either local variables, parameters, or objects are given imprecise method lattice types. We next discuss how our approach assigns more precise composite location types to each.

Local Variables

Within a single method, it is possible to introduce a superfluous cycle by assigning a method location type to each local variable (the default assignment strategy). Consider the following code excerpt:

```java
public void calculateIndex() {
    float f3 = c3 * this.curHum * this.curHum;
    ... 
    this.index = f3 + ...;
}
```

The method `calculateIndex` takes a value from the field `curHum`, computes it, then stores the result back to the field `index`. If we assign the composite location `<F3>` to the local variable `f3`, it will introduce a cycle in the method lattice as illustrated on the left side of Figure 5.6. The problem is that the assignment in Line 2 implies that the method location `<F3>` is lower than the method location `<THIS>`. By the same logic, the assignment in Line 4 would then create a cycle in the method location lattice. However, if we assign the variable `f3` to a composite location that begins with the method location `<THIS>`, e.g.
\langle \text{THIS, FRESH\_LOCATION} \rangle$, we can avoid introducing any cycles as illustrated on the right side of Figure 5.6.

![Figure 5.6: Superfluous Cycle](image)

**Parameters**

Interprocedural value flows can also introduce a superfluous cycle in the location types corresponding to value flows across method calls. For example, consider the following code excerpt:

```java
1 class foo {
2   int f, g;
3   void caller() {
4       int h = this.f;
5       callee(h);
6   }
7   void callee(int i) {
8       this.g = i;
9   }
```

The method `caller` takes a value from the field `f` in Line 4 and passes it as the argument to the method `callee` in Line 5, and then the method `callee` stores the value of the corresponding parameter `i` back to another field `g` of the same object in Line 8. Suppose that the location types of the current object `this` are \langle \text{THIS} \rangle for both methods. The assignment in Line 4 implies that the location of the variable `h` is lower than the location
(THIS, F). However, if we assign the method location ⟨LOCH⟩, which is lower than the location ⟨THIS⟩ in the method lattice, to the variable h without considering a field location type, there is a cycle—the value of the location ⟨LOCH⟩ eventually flows into the location of another field of the same object ⟨THIS, G⟩ in Line 8 through the method invocation.

To address this problem, our analysis summarizes the callee’s value flows that involve objects reachable from the caller’s arguments and transfer them to the caller so that we can remove the cycle in a later stage. When a callee has a value flow with an object that is reachable from a parameter, the analysis adds the corresponding value flows to the value flow graph of the caller in terms of the arguments. Our interprocedural analysis is structured as a fixed point computation. Whenever a method flow graph changes, all methods that call that method are scheduled for reanalysis.

**Objects**

It is possible to inadvertently create a cycle in the graph where a single flow traverses a set of composite locations that do not all have the same prefix (e.g., ⟨X, Y⟩ in ⟨X, Y, Z⟩).

Consider a location x in the value flow graph that (1) is reachable from a composite location y and (2) can reach a composite location z, where y and z share a common prefix that is not shared by x. The analysis constructs a new location that has the same prefix as y and z, followed by a fresh field location, and then assigns the new location to x.

To avoid a superfluous cycle, the process described here identifies one prefix that becomes the common prefix for all composite locations involved in the cycle. For example, in Figure 5.6, the algorithm identifies the common prefix THIS in the cycle, and then removes the cycle by assigning a new composite location with the prefix THIS to the location F3. However, it is possible for two objects to have two different cycles that each require assigning the other object to have the same prefix as a member of the first object. This
kind of cycle is not representable by the SJava type system. A graph may have a cycle with more than one common prefix. For such cycles, there is a choice of which object to select to become the common prefix. Since we have not seen this case appear in practice, the inference algorithm implementation simply selects at random a class to become the common prefix and prints a message for the developer. If such cases do appear, the implementation could be modified to try different common prefixes.

**Propagating Type Adjustments**

The resolution of cycles globally updates locations. So the prefix adjustments must be propagated throughout the value flow graphs. This process begins in the main event loop and proceeds downwards to each leaf method in the call graph. The composite location of a method call argument is propagated from the caller to the callee by translating the first element of the argument’s composite location into the callee context. For example, suppose there is a function call in which: (1) the argument has the composite location $\langle \text{OBJ}, A, B \rangle$, (2) the object whose method is invoked has the location $\langle \text{OBJ} \rangle$ in the caller, and (3) the callee defines the location of the current object `this` as $\langle \text{THIS} \rangle$.

In this case, the prefix $\langle \text{OBJ} \rangle$ in the caller is translated into the new prefix $\langle \text{THIS} \rangle$ in the callee, resulting in a composite location $\langle \text{THIS}, A, B \rangle$ for the method parameter within the callee.

**5.2.3 Inferring Program Counter Locations**

The program counter may be annotated with a location type, and if no annotation is provided, then it defaults to the top location. But this default assignment may be higher
than the program counter location at a method call. In this case the type checker will produce a type error at the call site.

Computing the lowest valid \( \text{PC} \) location provides the most flexibility as the method can be used whenever the caller’s program counter location is higher than the callee’s declared program counter location. Our tool first computes the set of parameter nodes \( P_m \) that have incoming flows in the flow graph. It then generates a program counter node \( \text{PC} \) satisfying the following constraint in the value flow graph.

**Program Counter Constraint:** For each parameter node \( p \in P_m \), there must exist an edge \((\text{PC} \rightarrow p)\) that indicates that the location type of the node \( \text{PC} \) must be higher than the location type of the parameter \( p \).

When one of the parameter locations \( p \in P_m \) is higher than all the others and it has a field location type, the node \( \text{PC} \) is assigned a new composite location with a field location type that is higher than the highest parameter node. If all the parameters have incoming flows, the tool elides the program counter annotation and simply relies on the default annotation.

### 5.2.4 Return Values

Annotating the return type of a method is necessary for completeness. When a program is type-checked, SJava computes the caller location for the return value that is consistent with the location types of the parameters and the return value. Therefore, computing the highest location for return values provides more flexibility to the caller context.

Our tool first computes the set of return value nodes \( r \in R \) such that there is a path \((p \leadsto r)\) in the value flow graph from a parameter node \( p \) to a return value node \( r \). Then, it generates a return location node \( \text{RLOC} \) satisfying the following constraint for return values.
Return Location Constraint: For each return value node \( r \in R \), there must exist an edge \((r \rightarrow \text{RLOC})\) that indicates that the location type of node \( \text{RLOC} \) must be lower than the location type of the return value node \( r \).

When one of return value nodes in the set \( R \) is lower than all of the others and it contains a field location type, the node \( \text{RLOC} \) is assigned a new composite location with a field location type relative to the lowest return value node, which provides more flexibility at call sites.

### 5.2.5 Hierarchy Graph

The goal of this phase in the algorithm is to transform the value flow graph into the method and field hierarchy graphs, each of which later is converted to a lattice suitable for producing annotations. In the value flow graph, information flows are not modeled in a modular manner — edges correspond to flows in both field and method lattices. Therefore, the inference algorithm decomposes the value flow graph into hierarchy graphs to capture information flows at the level of individual classes and methods, which makes the entire system composable.

We first classify edges in the value flow graph into two types of flows:

- **Method flows:** An edge represents a method flow if the first elements of two nodes are not identical. For example, the edge from \( n_1 = \langle v_1, f_m, \ldots, f_n \rangle \) to \( n_2 = \langle v_2, f_p, \ldots, f_q \rangle \) in the value flow graph represents a method flow from \( v_1 \) to \( v_2 \) in the corresponding method.

- **Field flows:** An edge represents a field flow if the prefixes of two nodes are identical and the subsequent fields are not identical. For example, the edge from \( n_1 = \langle v_1, \ldots, f_m, f_n \rangle \) to \( n_2 = \langle v_1, \ldots, f_m, f_o \rangle \) in the value flow graph represents a field flow from \( f_n \) to \( f_o \) in the corresponding class.
The inference algorithm next decomposes value flow graphs into two types of hierarchy graphs: a method hierarchy graph and a field hierarchy graph. For each method flow in the value flow graph, a corresponding edge is generated in the method hierarchy graph. For each field flow in the value flow graph, a corresponding edge is generated in the field hierarchy graph of the class containing the fields.

**Definition 2.** (Hierarchy Graph) A hierarchy graph is a directed graph $G = (H, E_h)$, where a node corresponds to either a field in the field hierarchy or a local variable in the method hierarchy, and an edge $(h_1 \rightarrow h_2) \in E_h$ denotes that there is either a field flow in the field hierarchy or a method flow in the method hierarchy from $h_1$ to $h_2$.

We present the basic algorithm for translating the value flow graph into method and field hierarchy graphs below:

- The algorithm first compares the source composite location $srcloc$ and destination composite location $dstloc$ and computes the index $idx$ of the first difference in the two composite locations. We will refer to the corresponding hierarchy as $diffhierarchy$.

- The algorithm next checks whether adding an edge from $srcloc[idx]$ to $dstloc[idx]$ in the hierarchy $diffhierarchy$ will create a cycle.

- If the addition would introduce a cycle, the algorithm merges all nodes in the cycle into a single shared location.

- Otherwise, the algorithm adds an edge from $srcloc[idx]$ to $dstloc[idx]$ in the hierarchy $diffhierarchy$.

The algorithm translates the edges in the value flow graph into the equivalent flow edges in either a method or field hierarchy. Step one of the algorithm identifies the first element of
the two composite locations that differ. Since the SJava type checker will evaluate the composite locations in lexical order, the algorithm must generate location flows that adhere to lexical ordering. Adding a flow in the corresponding hierarchy guarantees that the resulting location flows will type check. It is possible that adding the edge could introduce a cycle into a method or field hierarchy graph. If this occurs, the algorithm eliminates the cycle by merging all of the locations into a shared location. Figure 5.7 shows the method hierarchy graph for the `calculateIndex()` method and Figure 5.8 shows the field hierarchy graph for the `Weather` class.

**Figure 5.7:** Method Hierarchy

**Figure 5.8:** Field Hierarchy
5.2.6 Converting the Hierarchy Graphs into Lattices

At this point, the hierarchy graphs capture flows between all locations in the program. While the hierarchy graphs are partial orders, they may not be lattices as the GLB and LUB are not necessarily well defined. We may need to insert extra nodes into the partial order to make the GLB and LUB well defined. The problem of finding the smallest complete lattice that contains a partial order is known as the Dedekind-MacNeille completion. We use the algorithm developed by Nourine and Raynaud [38] to compute the Dedekind-MacNeille completion of the hierarchy graphs to generate lattices.

Figure 5.9 shows the inferred field lattice of the Weather class. For example, when there is a need to compare the combination of values from two location avgTemp and curHum with another value, the field lattice provides the GLB location LOC20 for two locations as the operand of the comparison.

5.2.7 Non-self-stabilizing Programs

A program can fail to self-stabilize if either (1) bad values remain at a location indefinitely or (2) bad values cycle indefinitely in a cyclic value flow. When values are not evicted from a location, the inference algorithm may infer location types that type check, but SJava’s static eviction analysis will reject the program. In the case of a cyclic value flow, it will attempt to use a shared location to eliminate the cycle. If the cycle contains either object references or fields of different objects, it will abort because it cannot be represented in the SJava type system. In this case, it provides value flow graphs to help developers fix the problems. For cycles that can be represented using shared types, it may potentially infer type annotations that type check. However, the stronger static eviction criteria required for shared locations will cause SJava’s static eviction analysis to reject the program.
Figure 5.9: Field Lattice

Figure 5.10: Simplified Field Lattice
5.2.8 Discussion

Our initial approach captured flow constraints using value flow graphs and hierarchy graphs, then produced lattices which are precisely compatible with the flow of values in the program. To maintain maximum precision, our algorithm created a unique mapping of each location type onto each node in the lattice. However, in practice, a program often defines a large number of variables and information paths. Therefore, maintaining maximally precise flow information may lead to a very complicated lattice that is incomprehensible and infeasible for developers to maintain.

Although the lattice for the example in Figure 5.9 does not look very complicated, it is a representation of value flows produced by the relatively small example program. After applying our initial implementation to the benchmarks, we found that the resulting lattices were too complicated for practical use. To give an idea of the complexity of maximally precise lattices, we show both the lattice structure for the `SynthesisFilter` class of the MP3 Decoder benchmark and a zoomed-in view for detail in Figure 5.11, which was generated by our initial implementation. It defines 997 locations types and has 10,491,169 possible information paths from the top to the bottom. This generated lattice is clearly incomprehensible to developers.

5.3 Simplification

The inference algorithm described in the previous section infers unique location types for every variable and field declaration, but often produces overly complicated lattices. However, generating simple lattices by reusing existing location types whenever possible can sometimes limit reusability. The reason is imprecise ordering relations between
Figure 5.11: Lattice for the SynthesisFilter
parameters and fields may require significant modifications when using the methods or fields in new contexts.

Our goal is to find a reasonable tradeoff between precisely modeling flow constraints and the simplicity of the resulting lattices. The idea is to maintain precise ordering relations between interface members (e.g., parameters, return values, program counters and fields) while reusing location types for non-interface members (e.g., local variables).

5.3.1 Constructing Interface Hierarchy Graphs

SJava optimizes for precise modeling of interface members by generating *interface hierarchy graphs* which only contain interface members. The process of simplification begins with removing non-interface nodes from the hierarchy graphs and then patching edges across the removals (potentially generating multiple edges for a removed edge). Returning to the example, the *Weather* class has four fields as interface members, shown as shaded nodes in Figure 5.8. Figure 5.13 shows the interface hierarchy graph for the *Weather* class.

5.3.2 Simplifying the Interface Hierarchy Graph

The algorithm next simplifies the interface hierarchy graphs by identifying nodes to merge. If two nodes have incoming edges that originate from the same set of nodes and outgoing edges that target the same set of nodes, then those two nodes can be merged while still maintaining the same precision (merging the nodes does not allow any new information flows). SJava identifies and merges such nodes.

It is possible for method and field hierarchies to contain redundant edges. If a hierarchy contains the edge $e = (n \rightarrow n')$ where $n'$ is reachable from $n$ through a path that does not
contain the edge $e$, then $e$ is redundant. The inference algorithm identifies and removes redundant edges in the hierarchies.

### 5.3.3 Inserting Merge Points

At this point, all nodes in the simplified hierarchy graph are either fields (for field hierarchies) or parameters, return values, and program counter locations (for method hierarchies). However, when the program combines value flows from more than one interface node, it needs a node in the hierarchy to store the combined flows. Consider the interface hierarchy graph shown in Figure 5.12. This hierarchy shows how information from the $a$, $b$, $c$, and $d$ fields are used to compute the values stored in the $f$ and $g$ fields. Such a computation could potentially begin by combining information from the $b$ and $c$ fields and storing it into a local variable. There is no location type in the hierarchy as shown that can be used for such a local variable. If we used the location type of either field $f$ or $g$, it would introduce a new flow into the hierarchy. Therefore, our tool must insert a merge point location type in the hierarchy that combines the information from fields $b$ and $c$ but is above both fields $f$ and $g$. The inference algorithm inserts a merge point whenever a non-interface node in the hierarchy graph combines incoming flows that originate from more than one interface node.

![Figure 5.12: Merge Point](image)

---
5.3.4 Converting the Interface Hierarchy Graphs into Lattices

The field and method hierarchy graphs currently capture the information flows between field and method interfaces, respectively. We compute the Dedekind-MacNeille completion of the field and method hierarchy graphs to generate field and method interface lattices, respectively.

5.3.5 Inserting Local Variable Locations

At this point, the interface lattices precisely capture information flows at the method or field interfaces. Now we insert locations for local variables into the lattices\(^2\). The idea is to splice in local variable nodes along existing edges in the interface lattice without changing the lattice’s meet or join structure. In the hierarchy graph, there could be more than one path consisting of local variable nodes between two interface nodes that are directly connected with each other in the interface hierarchy graph. These paths are comprised of two types of local variable nodes: normal nodes and shared nodes. The problem is to find the shortest sequence of nodes that contains every ordering sequence in a given set of paths.

\(^2\)Field lattices may contain locations for local variables with composite locations.
This problem is a generalization of the Shortest Common Supersequence problem\(^3\), which is known to be NP-complete [42]. We implemented the following approach that maps local nodes in the hierarchy graph to the location types in the lattices, heuristically minimizing the number of location types in the lattice.

- For a local variable node \(l\) in the hierarchy graph, find the lowest node \(m\) in the interface lattice that is above \(l\).
- Count the number of hops \(d\) between the local variable node \(l\) and the node \(m\). Each hop corresponds to a pair of a normal node and a shared node. The inference algorithm will insert either type of node to a pair if needed.
- We next convert the node \(m\) into a chain of nodes with the last node in the chain having \(m\)’s outgoing edges. The inference algorithm first checks if there already exists the \(d\)th hop pair along this chain. If such a pair does not exist or the same type of node does not exist in the pair, we insert a new node. Otherwise, we reuse the existing node.

The inference algorithm generates a final lattice that admits more flows between local variables than the actual program performs, but suffices to show that the program self-stabilizes.

Figure 5.14 illustrates how our approach optimizes the hierarchy graphs. In the figure, the shaded circles represent the interface nodes and the dotted circles represent the local variables. First, the analysis simplifies the graph by merging two interface nodes \(f\) and \(g\) since both nodes share the same incoming and outgoing edges in the interface hierarchy graph. Then, it generates the lattice in which local variables share locations. Figure 5.10

\(^3\)A shortest common supersequence problem can be represented as our minimization problem by creating a unique path for each input sequence, encoding 0 with a normal node and 1 with a shared node. Thus finding an optimal solution to our problem is at least as difficult as the shortest common supersequence problem.
Figure 5.14: *Hierarchy Graph and Lattice*

presents the inferred lattices for the example, which is much simpler than the non-optimized lattice in Figure 5.9. In the lattice, local variables with the same height are assigned to the same node, and non-interface members (white nodes) are arranged in a simple structure between interface members (shaded nodes) while the optimization maintains precise orderings between interface members.

Figure 5.15 shows how each location type in the lattices maps to the concrete memory location by using SJava annotations in the example code.
public class Weather {
  @LATTICE("index<Loc4,...,Loc6<avgTemp,
  avgTemp<Loc11,Loc11<prevTemp,Loc3<curHum")
  public float prevTemp;
  @LOC("avgTemp") public float avgTemp;
  @LOC("avgTemp") public float curHum;
  @LOC("index") public float index;

  // define constants c1 to c9
  public static final float c1 = -0.22475541;

  @LATTICE("this<inTemp")
  @THISLOC("this")
  public void calculateIndex(){
    while(true) {
      @LOC("inTemp") float inTemp = Device.readTemp();
      curHum = Device.readHumidity();
      // calculate the average temperature
      avgTemp = (prevTemp+inTemp)/2;
      prevTemp = inTemp;
      float f1=c1*avgTemp*curHum;
      float f2=c2*avgTemp*avgTemp;
      float f3=c3*curHum*curHum;
      float f4=c4*f2*curHum;
      float f5=c5*f3*avgTemp;
      float f6=c6*f1*f2;
      index = c7+c8*avgTemp+c9*curHum+
        f1+f2+f3+f4+f5+f6;
      broadcast(convertToString(index));
    }
  }
}

Figure 5.15: Weather Index Example with Inferred Annotations
Chapter 6

Evaluation

We have implemented our inference tool in the SJava compiler and evaluated it on an Ubuntu Linux 12.04 machine with an Intel Core i7 3770 CPU. We have evaluated our tool by using three existing Java applications: JLayer, an MP3 decoder; LEA, an eye-tracker; and Sumo Robot, a robot controller.

Section 6.1 describes three benchmarks. Section 6.2 presents an experimental evaluation of the self-stabilization. In Section 6.3, we evaluated our inference algorithm in terms of correctness and simplification goals.

6.1 Benchmarks

We selected three self-stabilizing programs that are structured with the main event loop.

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.

LEA is a lightweight eye tracking algorithm library and is available at [http://sourceforge.net/projects/lea-eyetracking/](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.

Sumo Robot controls robots and is available at [http://java.net/projects/sumorobots/](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.

6.2 Self-Stabilization Evaluation

In this section, we first annotated source code, and then executed our tool to check if the annotated program self-stabilizes. We also 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.

6.2.1 MP3 Decoder

We use our tool to automatically check that the 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 6.1 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 6.2 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.

6.2.2 Eye Tracking

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.

6.2.3 Robot Control

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.
6.2.4 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 6.3 summarizes our annotation effort. For each benchmark, we list the number of location assignments using \texttt{@LOC} in the Location column, the number of lattice definitions using \texttt{@LATTICE} in the Lattice column, the number of default method lattice definitions using \texttt{@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.
### Table 6.1: Evaluation Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Simple(&lt;=5)</th>
<th>Complex(&gt;5)</th>
<th>Time</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Locations</td>
<td>Paths</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MP3</td>
<td>manual</td>
<td>141</td>
<td>35</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>naive</td>
<td>176</td>
<td>61</td>
<td>1,998</td>
</tr>
<tr>
<td></td>
<td>SInfer</td>
<td>205</td>
<td>62</td>
<td>421</td>
</tr>
<tr>
<td>Eye</td>
<td>manual</td>
<td>215</td>
<td>59</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>naive</td>
<td>183</td>
<td>58</td>
<td>503</td>
</tr>
<tr>
<td></td>
<td>SInfer</td>
<td>161</td>
<td>67</td>
<td>343</td>
</tr>
<tr>
<td>Robot</td>
<td>manual</td>
<td>132</td>
<td>36</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>naive</td>
<td>149</td>
<td>44</td>
<td>161</td>
</tr>
<tr>
<td></td>
<td>SInfer</td>
<td>161</td>
<td>45</td>
<td>79</td>
</tr>
</tbody>
</table>

#### 6.3 Inference Evaluation

##### 6.3.1 Methodology

We evaluated the inference algorithm by taking the source code for the three benchmarks and inferring annotations. As the original applications required minor changes to make them self-stabilizing, we took the modified versions of the SJava benchmark and removed all of the location type annotations.

**Correctness** We used the SJava type checker to verify the correctness of the generated annotations. All three benchmarks type check and pass SJava’s eviction analysis, and thus they are self-stabilizing.

**Simplification Goals** Our evaluation was designed to evaluate the usefulness and understandability of the inferred lattices. Even though such evaluations depend on subjective criteria and how developers use the results, the complexity of the lattices is an important factor in evaluating the usefulness and understandability. Therefore, we developed two metrics to measure the complexity of the lattices. First, we approximated the complexity by comparing the total number of locations in the lattices between SInfer and the naïve approach that attempts to maintain maximal precision described in
Section 5.2. However, measuring the complexity is more complicated than comparing the total number of locations. For example, if a large number of locations are arranged into a single line, this structure could be more easily understandable than a complex structure with a smaller number of locations. Therefore, we defined another quantitative measurement, the number of paths from the top to the bottom in a lattice, which captures the number of different ways values flow through a lattice. McCabe [35] developed a similar metric to measure program complexity.

Table 6.1 provides information about the lattices generated by the naïve approach and SInfer. It also shows information about the manual annotations. Even though SInfer generates a smaller number of locations and paths than the naïve approach, we found that the total numbers tend to be biased by many simple lattices. Therefore, we split lattices into two categories: simple lattices and complex lattices with complex lattices defined as having more than 5 nodes. The threshold columns labeled <= 5 and > 5 show the total numbers of the simple lattices and the complex lattices respectively. Note that, in some cases, the total number of SInfer lattices in the simple category is larger than the total number of the naïve approach because SInfer is able to simplify complicated lattices and place them in the simple category. The last two columns show the time for type inference and lines of code for each benchmark. SInfer is slower than the naïve approach because the former requires an additional process for the simplification.

### 6.3.2 MP3 Decoder

Of the three benchmark applications, the MP3 decoder had the highest potential for complicated annotations. Many methods in the program employ a processing pattern which first loads data into a set of source fields, then transforms the data in a sequential computation using several local variables, and finally store the results into a set of
destination fields. The problem is that the multiple stages of computation that extensively use temporal variables to store intermediate results create a large number of value flows. As shown in Figure 5.11 in Section 5.2.8, our naïve approach, which attempted to preserve as much precision as possible in the value flow graph, generated an incomprehensible lattice.

To quantify how well we have met the simplification goal, we present results comparing SInfer to the manual annotations and to the naïve approach in Table 6.1. It is clear that the new strategy helps effectively reduce the number of locations and paths. For the MP3 decoder, SInfer generated 421 location types and 542 paths for complex lattices, whereas the naïve approach generated a total of 1,998 location types and 294,624,128 paths. SInfer infers slightly more complicated annotations than the manual annotations. The manual approach used some tricks to reduce location types — the manual annotations used shared location types simply to avoid generating a chain of location types in a lattice. While this approach does reduce the number of annotations that must be written, it can also be misleading as it may lead new developers to believe that a cyclic value flow exists where it does not. Thus although the automatic annotations contain more location types, they may in fact be preferable for program understanding.

Ideally, SInfer will generate annotations that the developer can intuitively understand. The MP3 decoder is our best benchmark for evaluating the readability of the generated annotations, because it is the most complex. We manually examined the generated lattices and found that their structures clearly show the flow of values through the program. As an example, Figure 6.4 shows the inferred lattice of the SynthesisFilter class. In the lattice, the location types assigned to the fields outlined a distinct hierarchy, and it was easy to correlate each level of that hierarchy with a phase of the sequential decoding process.

Moreover, qualitatively, the automatically inferred lattices had significant structure — it was easy to visually extract the flow of information. Most location types were structured in
simple linear chains. This provides some evidence that the tool may be useful for program understanding and debugging.

Figure 6.4: The Lattice of Synthesis Filter
6.3.3 Eye Tracking

The original manual annotations identified an opportunity to avoid introducing a large number of composite locations: the part of the computation that detects eye positions could use the same location for all eye position fields. Our inference tool identifies the same opportunity to simplify the lattice, and generates lattices that are straightforward to understand.

6.3.4 Robot Controller

The program follows a common pattern that is likely shared by a broad range of embedded controllers: at every iteration, such a controller takes a new input, processes it, overwrites all execution-specific memory locations, and emits a result. This pattern results in a straightforward location type hierarchy, because (1) each computation stage directly matches one level of the type hierarchy and (2) Sumo Robot does not rely heavily on the use of objects, which would complicate the hierarchy.
Chapter 7

Related Work

In this chapter, we first discuss existing approaches to robust software systems in Section 7.1. In Section 7.2, we address related work on self-stabilization for fault tolerance. Lastly, we survey related work on annotation inference techniques in Section 7.3.

7.1 Approaches to Robust Software Systems

Researchers in academic and industry have developed several different approaches to creating robust software systems. Despite concerted efforts from the research community, however, developing robust software systems continues to pose challenging, wide-open problems. In what follows, we summarize related work in the following three areas.

- **Eliminating Failures**: An approach designed to prevent software faults from occurring.
- **Failure Recovery**: An approach that enables software systems to survive upon a software fault.
• **Ignoring Failures**: An approach that keeps software systems operating through errors.

### 7.1.1 Eliminating Failures

Since type checking [3] enforces very simple safety properties, developers use it to statically verify the absence of certain types of runtime bugs. However, type checking cannot express temporal and context-dependent constraints. In response, typestate checking [50, 37] was designed to detect any non-sensical operation in the current context (e.g., a file should not be read after being closed). It guarantees that programs do not violate certain kinds of temporal safety properties.

In numerous past efforts to detect bugs, static analysis has been proven useful [21, 29, 26]. It not only requires merely a shallow understanding of code, but also enables users to locate as many bugs as possible. At the same time, static analysis cannot prove the absence of bugs, but only finds some possible bugs. Moreover, many approaches depend on a predefined pattern of the bug.

Researchers have proposed formal verification as an approach to proving the absence of software bugs. One approach is model checking [28, 6, 53, 4], which involves constructing a model of the system and performing an exhaustive search to verify the absence of any path from an initial state to an error state. However, model checking poses a fundamental problem, for users must always deal with immense state space. It is also difficult to obtain a model of the system that is as good as the actual implementation.

Another instance of formal verification is theorem proving [39], in which the behavior of the system and its desired properties are first expressed in logic. Next, the user applies a
theorem prover to verify its correctness. Here, however, the problem is that the result of verification heavily relies on the user’s understanding and expertise of the system.

In the software industry, practitioners depend on software testing that involves a process of executing software systems with the goal of finding errors [5]. Yet, this testing approach cannot catch all bugs and often requires human intervention [7].

A critical difference among these approaches and ours is that we eliminate the need to precisely define the correct behavior or type of errors. While these approaches help developers to find bugs in advance, they do not address transient hardware errors.

7.1.2 Failure Recovery

Recovery techniques attempt to correct the system when it enters into an erroneous state. For one, upon failures, backward recovery [12, 54, 40] tries to rollback to the most recent checkpoint and then re-execute the system upon failures. By contrast, forward recovery [30] attempts recover by finding a new state in which the system can continue operation. Upon failures, it compensates error by having redundancy to derive the correct answer.

Design diversity [1, 2] executes the set of implementation versions independently developed from the same specification. The problem with these approaches, however, is that they inevitably require more effort and budget to develop multiple independent implementations.

Alternatively, restarting [11, 31] provides a basic approach to failure recovery. When a software system enters into an erroneous state, it attempts to restart the system with in order to recover a wide range of failures. However, this methodology cannot be applied to a system that cannot tolerate the loss of all states.
Compared to these recovery techniques, the pivotal advantage of SJava is that it does not require recovery routines. Writing recovery routines often relies on a specific failure assumption, and can thus lead to another failure if the error state differs from the assumption. Instead of having an error-prone recovery routine, our approach continues operating the software system and allows it to recover its normal behavior when an erroneous state is resolved.

### 7.1.3 Ignoring Failures

Failure-oblivious computing [43] enables programs to continue execution beyond 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 has detected bugs and attempted re-execution in a slightly different environment [47]. Data structure repair [15] adopts an interventional approach; upon detecting data structure corruption, it repairs such corruption with respect to a specification. Data structure repair only guarantees only that a program will reach a consistent state, whereas, our work guarantees that all effects of the bug eventually disappear. Moreover, since our approach does not require a specification, it eliminates the need to precisely define correct behavior.

### 7.2 Self-Stabilization for Fault Tolerance

Self-stabilization was initially suggested by Dijkstra in the context of robust distributed algorithms [16]. Recent work by Dolev et al. [18, 19, 17] that proposes techniques to ensure that the underlying layers (e.g., processor, operating system, and compiler) preserve the self-stabilizing nature of an application. Yet, 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.

7.3 Language-based Information Flow and Inference

Language-based information flow employs type systems to check that information flows in an application do not violate desired requirements [36, 45, 46]. Our approach is similar in some aspects. However, the critical differences are that our approach must support much finer-grained divisions of data, as well as check that values only remain in a given location for a bounded time.

We believe that our work in inferring flow annotations can be adapted for other information flow type systems. For inferring secure information flow, numerous approaches have been proposed for secure information flow inference. Smith et al. [48] and King et al. [33] have attempted to reduce the annotation burden, though their methods still require developers to define security policies. By contrast, our system automatically infers all specifications and type annotations. Vaughan et al. [51] have presented a security policy inference tool that does not infer method annotations, while Livshits et al. [34] have developed a probabilistic inference tool for explicit information flow. However, this tool infers only information flows in terms of the methods involved, which is too coarse for the SJava type system. The constraint-based type inference system [41] has a different goal from our system: while it seeks to infer types as precisely as possible, ours infers location types that meet our simplicity goal at the expense of some precision.

Several tools have been developed to infer specifications. A prominent example is Daikon [24], which infers specifications for program variables and fields. Other tools have been used to automatically infer Java generics annotations [20] and concurrency...
specifications [10]. Though our work shares the high-level goal of alleviating the manual annotation labor, the techniques and goals of these tools differ.
Chapter 8

Conclusion and Future Work

Self-stabilization has been used as an approach for building robust distributed systems. 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. In certain important domains, self-stabilization can improve the robustness of software applications. Moreover, self-stabilization is complimentary to existing approaches for improving reliability.

A developer simply annotates the source code either from scratch or by using the annotation inference to capture the flow of values. The SJava compiler then checks that incorrect values will eventually leave the program, thereby returning the program to the exact correct state. Our experience indicates that this approach can successfully check the self-stabilization of our benchmark applications.

There are numerous possible approaches for extending SJava:
• **Hybrid Self-stabilizing System:** Clearly, all programs are not completely self-stabilizing by nature. We observe, however, that many of their sub-components are.

The goal of our hybrid system is to provide a way to coexist with non-self-stabilizing components. For this, we can divide the states of a system into two components: the self-stabilizing component and the user-managed stateful component. If the states in the user-managed stateful component are strongly connected and the user can control all possible transitions between states with a sequence of user inputs, then the proper user input that makes the system transition from an incorrect state to correct state in the user-managed stateful component and the self-stabilizing component can make the overall system self-stabilizing. SJava checker combined with additional annotations and static analysis, which help the compiler to analyze state transitions in the stateful component, can thereby check whether the input program has such properties.

• **Handling Hardware Faults:** As discussed in Section 1.1.2, SJava can handle certain types of transient hardware errors. Though it remains impossible to tolerate all possible hardware errors by using software-based techniques, SJava can be further extended to make a program highly resistant to hardware errors. The idea is to generate code that employs redundancy and the error correcting mechanism to tolerate hardware errors until the hardware starts to work correctly again.

• **Supporting Other Languages:** Our general approach is not limited to Java language. We believe that our approach can be easily applied to other programming languages that provide at least strong type safety (e.g., C dialects [32]). We can therefore implement our approach based on a dialect of C to support a wide range of embedded applications.
• **Memory Management**: SJava can easily be extended to avoid garbage collection inside the main event loop. The properties checked by the current analysis imply that all objects allocated in the main event loop are eventually not accessed in the future. A simple analysis of the lattice can produce symbolic bounds on the lifetime of such objects.
Bibliography


Enerj: Approximate data types for safe and general low-power computation. In
Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language
ACM.

immune system for software services. In Proceedings of the Annual Conference on
USENIX Annual Technical Conference, ATEC ’05, pages 11–11, Berkeley, CA, USA,

In Proceedings of the 2007 Workshop on Programming Languages and Analysis for

[49] F. Spoto, F. Mesnard, and E. Payet. A termination analyzer for java bytecode based


Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP ’11, pages

[52] P. Wadler. Linear types can change the world! In PROGRAMMING CONCEPTS
AND METHODS. North, 1990.

serious file system errors. In Proceedings of the 6th Conference on Symposium on
Operating Systems Design & Implementation - Volume 6, OSDI’04, pages 19–19,