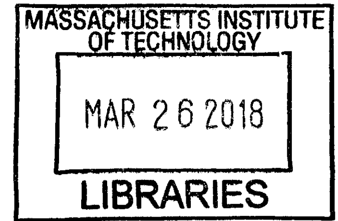
Leto: Verifying Application-Specific Fault Tolerance via First-Class Execution Models



ARCHIVES

by

Brett Boston

B.S., University of Washington (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Author . . Signature redacted . . . .
Department of Electrical Engineering and Computer Science
December 6, 2017

Certified by . . . . Signature redacted . . . .
Michael Carbin
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . Signature redacted . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Leto: Verifying Application-Specific Fault Tolerance via First-Class Execution Models

by

Brett Boston

Submitted to the Department of Electrical Engineering and Computer Science
on December 6, 2017, in partial fulfillment of the
requirements for the degree of
Masters of Science in Electrical Engineering and Computer Science

Abstract

Due to the aggressive scaling of technology sizes in modern computer processor fabrication, modern processors have become less reliable andmore prone to exposing hardware errors to software. In response, researchers have recently designed a number of application-specific fault tolerance mechanisms that enable applications to either be naturally resilient to errors or include additional detection and correction steps that can bring the overall execution of an application back into an envelope for which an acceptable execution is eventually guaranteed. A major challenge to building an application that leverages these mechanisms, however, is to verify that the implementation satisfies the basic invariants that these mechanisms require – given a model of how faults may manifest during the application's execution.

To this end I present Leto, a verification system that enables developers to verify their applications with respect to a *first-class* execution model specification. Namely, Leto enables software and platform developers to programmatically specify the execution semantics of the underlying hardware system as well as verify assertions about the behavior of the application's resulting execution.

A key aspect of verifying these implementations is that applications leveraging application-specific fault tolerance mechanisms often require assertions that relate the behavior of the implementation's execution in the presence of errors to a fault-free execution. To support this, Leto specifically supports *relational* verification in that its assertion language enables a developer to specify and verify assertions that relate the two semantics of the program.

In this thesis, I present the Leto programming language and its corresponding verification system. I also demonstrate Leto on several applications that leverage application-specific fault tolerance mechanisms.

Thesis Supervisor: Michael Carbin
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgements

First and foremost I would like to thank my advisor, Professor Michael Carbin, for his guidance and patience in the completion of the research that went into this thesis. I look forward to our continued collaboration as I begin my PhD research.

Thanks to the rest of the Programming Systems Group for their insightful research discussions, and for tolerating way too many of my cryptocurrency comments.

Thanks to Professor Leslie Kolodziejski and Janet Fischer for being understanding and supportive when I fell ill during the completion of this thesis.

Thanks my parents, Roya Sohaey and David Boston, as well as my sister, Haley Boston, for their unconditional love and support. Thanks for recognizing and supporting my childhood interest in computers, I wouldn't have made it this far without you.

Thanks to my many friends at MIT, including Thomas Bourgeat, Max Dunitz, Sara Achour, Peter Ahrens, Jack Feser, Jarrett Revels, Nick Bandiera, and many others for making grad school such an enjoyable place.

Thanks to the many volunteers at WMBR for providing me with an avenue to express myself.

Lastly, I would like to thank the greater CSAIL community for reading and interacting with my silly weekly GSB emails. Extra thanks to those who attended GSB and made the event so much fun to host.

This research was supported in part by the United States Department of Energy (Grants DE-SC0008923 and DE-SC0014204).

Contents

1 Introduction **11**

1.1 Traditional Fault Tolerance 13

1.2 Application-Specific Fault Tolerance 14

1.3 Verifying Application-Specific Fault Tolerance 15

1.4 Contributions 18

2 Example **21**

2.1 Jacobi Implementation 23

2.2 Specification. 24

2.3 Verification Approach 26

2.4 Verification Algorithm 29

3 Language **33**

3.1 Dynamic Semantics 36

3.1.1 Preliminaries 36

3.1.2 Execution Model Semantics 38

3.1.3 Language Semantics 39

4 Logic **43**

4.1 Preliminaries 43

4.2 Proof Rules 48

4.2.1 Left and Right Rules for Primitive Statements 48

4.2.2 Left and Right Rules for Control Flow 50

4.2.3 Lockstep Rules for Control Flow . . . 51
4.3 Properties . . . 52

5 Verification Algorithm **61**
5.1 Invariant Inference . . . 66
5.2 Implementation . . . 72

6 Case Studies **73**
6.1 Benchmarks and Properties . . . 73
6.2 Verification Effort . . . 74
6.3 Runtime Characteristics . . . 75
6.4 Execution Models . . . 76

7 Self-correcting Connected Components **79**
7.1 SC-CC Implementation . . . 82
7.2 Specification . . . 86
7.3 Verification Approach . . . 88

8 Self-stabilizing Conjugate Gradient Descent **93**
8.1 Implementation . . . 96
8.1.1 Reliable Correction Step . . . 96
8.1.2 Faulty Matrix Vector Product . . . 97
8.2 Specification . . . 98
8.3 Verification Approach . . . 99
8.3.1 Correction Step . . . 99
8.3.2 Faulty Matrix Vector Product . . . 100

9 Self-Stabilizing Steepest Descent Correction Step **103**
9.1 SS-SD Correction Step Implementation . . . 104
9.2 Specification . . . 106
9.3 Verification Approach . . . 107

10 Related Work **109**

11 Conclusion **113**

A Full Semantics **123**

B Full Self-stabilizing Conjugate Gradient Descent Implementation **127**

Chapter 1

Introduction

Due to the aggressive scaling of technology sizes in modern computer processor fabrication, modern processors have become more vulnerable to errors that result from natural variations in processor manufacturing, natural variations in transistor reliability as processors physically age over time, and natural variations in these processors' operating environments (e.g., temperature variation and cosmic/environmental radiation) [6].

Large distributed systems composed of these processors – such as emerging designs for exascale supercomputers – are anticipated to encounter errors frequently enough that traditional techniques for building high-reliability applications will be too resource intensive (both in time, storage, and energy-consumption) to be practical. Applications will, instead, need to be architected to execute through errors [1].

In pursuit of increased performance, increased scalability, and increased ubiquity for computation, emerging computational models are pushing the boundaries of the reliable, digital computation abstraction.

Scaling Traditional Processors. Due to the aggressive scaling of technology sizes in modern computer processor fabrication, modern processors have become more vulnerable to errors that result from natural variations in processor manufacturing, natural variations in transistor reliability as processors physically age over time, and natural variations in these processors' operating environments (e.g., temperature variation and cosmic/environmental radiation) [1, 6, 23, 27, 37–39, 57, 69].

Large-Scale Distributed Computations. Large distributed systems – such as emerging designs for exascale supercomputers – are anticipated to encounter errors frequently enough that traditional techniques for building high-reliability applications will be too resource intensive (both in time, storage, and energy-consumption) to be practical [1, 59].

Applications will, instead, need to execute through errors. In addition, emerging computations, such as stochastic gradient descent-based learning techniques, have evolved into asynchronous, racy implementations in their quest for high-performance in environments where the communication overhead of traditional, synchronous versions of these algorithms would otherwise limit the scalability of these techniques [48].

Energy Harvesting Systems. Researchers have recently proposed small, batteryless computing platforms that operate entirely on energy harvested from their environments (e.g., solar, RF, and kinetic) [18, 42]. A key challenge with these systems is that due to the lack of a battery, these systems may spontaneously shutdown when they exhaust their available energy. The execution model of these systems is therefore *intermittent* in that computing resources may disappear at any time and, therefore, application developers need to manage concerns such as unanticipated control flows that restart the application and corrupt the application's state, as well as checkpointing and recovery to manage the consistency of data across these control flows.

Cyber-Physical Systems. Systems that operate on the border of the digital world and the physical world, such as autonomous agents, typically also receive inputs from physical components or control physical outputs that may be subject to uncertainty in their results as well as physical failures.

Researchers are investing significant energy in developing methodologies to build systems that are safe and robust in the presence of uncertainty and failures.

Challenges. All of these systems encounter *faults* – anomalies in the underlying physical device – that produce *errors* – unanticipated or incorrect values that are visible to the application. Simple error models include bit flips in the output of arithmetic, logical, and memory operations. A key challenge for building applications for these platforms is that

reasoning about the *reliability* of these applications requires reasoning about the operation of the underlying execution model and its impact on the application's behavior.

1.1 Traditional Fault Tolerance

Researchers have long sought methods to enable reliable computation on unreliable computing substrates. For example, in the 1950s, vacuum-tube-based computing systems experienced vacuum-tube failures as frequently as every 8 hours [66]. In response, the industrial and academic community sought to resolve this issue by both:

- Designing more reliable computing substrates (modern CMOS transistors).
- Designing fault tolerance mechanisms.

Of these latter techniques, popular methods include

- **Dual-Modular Redundancy (DMR)**. DMR enables error detection by duplicating instructions and verifying that the results of the duplicated computation agree with each other [49].
- **n-Modular Redundancy**. Like DMR, this technique executes instructions multiple times. However, when $n > 2$ it also enables error correction in addition to error detection by using the result agreed upon by a majority of the replicated instructions [58].
- **Algorithm-Based Fault Tolerance (ABFT)**. ABFT methods are modifications to algorithms to allow them to detect and/or correct for errors encountered in computation. This technique is application-specific and may provide lower-overhead detection and correction than n-modular redudancy schemes [21, 55].

A major aspect of the design of such mechanisms is the trade-off between the overhead (in performance, memory consumption, and energy-consumption) of these techniques, the frequency and distribution of hardware faults, and the coverage of a specific error detection and correction scheme. For example, standard methods for dual-modular redundancy duplicate

the entire execution of a computation and check if the two executions of the computation agree on their results. This technique introduces significant computational and energy overhead. In contrast, algorithm-based fault tolerance techniques – such as those for linear algebra – produce lightweight checksums that can be used to validate if the computation produced the correct results. For some applications, these checksums are exact, enabling the exact error detection capabilities of dual-modular redundancy but with lower overhead. However, for other applications, these checksums either are not known to exist or, at best, compromise on their error coverage.

1.2 Application-Specific Fault Tolerance

Modern large computing systems have begun to operate a point in the trade-off space between performance/energy and error rates that traditional, application-oblivious fault tolerance techniques are too resource intensive to deploy at scale for large numerical computations [1].

In response, researchers have begun to expand on historical results for algorithm-based fault tolerance [21, 35], alternatively *application-specific fault tolerance*, to identify new opportunities for low-overhead mechanisms that can steer an application's execution to produce *acceptable* results: results that are within some tolerance of the result expected from a fully reliable execution.

Such techniques include selective n-modular redundancy in which a developer either manually – or with the support of a fault-injection tool – identifies instructions or regions of code that do not need to be protected for the application to produce an acceptable result – as determined by an empirical evaluation [14, 62, 64, 65]. Another class of techniques are fault-tolerant algorithms that through the addition of algorithm-specific checking and correction code are tolerant to faults [16, 20, 54, 55]. A major barrier to implementing either of these techniques is that their results either rely on empirical guarantees or – for self-stabilizing algorithms – hinge on the assumption that the fault model of the underlying computing substrate matches the modeling assumptions of the algorithmic formalization.

1.3 Verifying Application-Specific Fault Tolerance

To address these challenges I present Leto, a verification system that supports reasoning about unreliably executed programs. Leto enables a developer to build confidence in their application-specific fault tolerance mechanism by:

- Enabling a developer to programmatically specify the behavior of the computing substrate's fault model.
- Enabling a developer to verify *relational* assertions that relate the behavior of the unreliably executed program to that of a reliable execution.

Namely, Leto enables a developer to specify the behavior of the underlying hardware system as a program that Leto automatically weaves into the execution of the main program. In addition, Leto enables a developer to specify relational assertions that, for example, constrain the difference in results of the unreliable execution of the program from that of its reliable execution.

First-Class Execution Models. Leto enables a developer to programmatically specify a stateful execution model. For example, a common fault model that application developers use is the *single-event-upset* model. In this model, at most one fault can occur during the execution of the program. While simple, this model can capture real fault models in which it is possible for errors to happen during execution, but with small probability. Figure 1-1 presents a specification in Leto of a single-event upset model that affects only addition and multiplication operations.

Model State. An execution model may include state in the form of variables that the model may use to implement its operations. In this case, the model includes the boolean variable `upset` which – if set to `true` – indicates the model has produced a fault and will no longer produce any faults during the execution of the program.

Operations. The execution model specification includes a specification of the behavior of each operation that it exports to an application. An operation includes the name of

```
1  const real E = ...;
2  bool upset = false;
3
4  operator +(x1, x2)
5      modifies ()
6      ensures (result == x1+x2);
7  operator +(x1, x2)
8      when (!upset)
9      modifies (upset)
10     ensures (x1 + x2 - E < result < x1 + x2 + E && upset);
11
12 operator *(x1, x2)
13     modifies ()
14     ensures (result == x1*x2);
15 operator *(x1, x2)
16     when (!upset)
17     modifies (upset)
18     ensures (x1 * x2 - E < result < x1 * x2 + E && upset);
```

Figure 1-1: Single Event Upset Error Model

the operation, the arguments to the operation, as well as the operation's *guards*. A single operation may have multiple specifications. For example, the model specification includes two specifications for the addition operator, +. The first specification on Line 4 specifies the *reliable*, standard implementation of addition. This specification specifies the operation as a function `operator +(x1, x2)` that takes as input the left and right operands of the addition, `x1` and `x2`, respectively. The specification declaratively specifies the semantics of the operation with an `ensures` clause that specifies the relation between the distinguished result of the operation, `result`, and the operation's inputs: `ensures result == x1 + x2`.

The second specification on Line 8 specifies a *relaxed* implementation that may expose faults to the application. The specification specifies the operation similarly as a function `operator +(x1, x2)`. In contrast to the reliable version of the operation, this specification includes a `when` clause that guards execution of this implementation. In this case, the `when upset` specifies that this operation specification is only enabled if no fault has previously occurred. This guard therefore enforces the SEU design of this model. If the guard is satisfied, then the operation returns a result that satisfies

```
ensures x1 + x2 - E < result < x1 + x2 + E && upset.
```

This ensures clause bounds the result of the addition to be within `E` of the actual sum. In addition, the operation sets `upset` to `true` indicating that the single fault for the model has occurred. Because Leto's execution model specification language is general (with no restrictions on model state or how it is manipulated by the operation specifications), developers and platform-designers can specify rich execution models for a given platform. In Chapter 6 I provide additional execution model specifications.

Semantics. For every operation to be executed, the execution non-deterministically chooses among the enabled implementations (given each implementation's guard condition) and uses the chosen implementation's specification to update the state of the program.

This model includes a boolean valued state variable that records whether or not a fault has occurred during the execution of the program. The model then exports two versions of the addition and multiplication operator. Line 4 specifies the reliable implementation of addition while Line 8 specifies an unreliable version. For each addition operation in a program, Leto dynamically makes a non-deterministic choice between the set of operation implementations that are currently *enabled* in the model. An operation is enabled if its *guard* evaluates to *true*. Namely, an operation's guard is the optionally-specified boolean expression that occurs after the `when` keyword. For these two versions of addition, the reliable version is always enabled and the unreliable version is enabled only if `!upset` – indicating that a fault has yet to occur in the program.

Relational Verification. Verification of an application that has been protected with an application-specific fault tolerance mechanism typically requires reasoning about two types of properties of the resulting application: *safety properties* and *accuracy* properties.

- **Safety Properties:** safety properties are standard properties of the execution of the application that must be true of a single execution of the application. Such properties include, for example, memory safety and the assertion that the application returns results that are within some range. For example, a computation that computes a distance metric must – regardless of the accuracy of its results – return a value that is non-negative. In Leto, a developer specifies safety properties with the standard `assert`

statements as typically seen in verification systems.

- **Accuracy Properties:** accuracy properties are properties of the unreliable or *relaxed* execution of the application that relate its behavior and results to that of a reliably executed version. Accuracy properties are *relational* in that they relate values of the state of the program between its two semantic interpretations. For example, the assertion `-epsilon < x<o> - x<r> < epsilon` in Leto specifies that the difference in value of `x` between the program's reliable execution (denoted by `x<o>`) and relaxed execution (denoted by `x<r>`) is at most `epsilon`.

Leto provides and implements a Relational Hoare Logic [4, 5, 12, 60] as its core program logic. Relational Hoare Logics are a variant of the standard Hoare Logic that natively refer to the values of variables between two executions of the program. Leto's use of a relational program logic serves two goals:

- It gives a semantics to accuracy properties.
- It enables tractable verification of safety properties.

For example, proving the memory safety of an application outright can be challenging for many applications. However, application-specific fault tolerance mechanisms can typically be designed and deployed such that it is possible to verify that for any given array access or memory access, errors in the application do not *interfere* with the accessed address. Such properties are typically easier to verify for a protected application than verifying the safety of the memory access outright. Leto therefore enables developers to tractably verify a strong *relative* safety guarantee: if the original application satisfies all of the specified safety properties, then relaxed executions of the application with its deployed application-specific fault tolerance mechanisms also satisfy these safety properties. By contraposition, if a relaxed execution violates a safety property, then the original application also violates a safety property.

1.4 Contributions

This thesis presents the following contributions:

- **Language for Execution Models:** I present a declarative language for execution model specifications that provides guarded non-deterministic selection of each operation's implementation.

- **First Class Execution Modeling:** I present language constructs that enable the developer to specify assertions that reference the state of the execution model. These constructs enable a developer to, for example, specify the precise behavior that is required to verify high-level convergence properties for self-stabilizing applications.

- **Programming Language and Semantics:** I present a programming language and semantics that enables a developer to verify the validity of both standard assertions (safety properties) and relational assertions (accuracy properties).

- **Verification Algorithm:** Leto includes a verification algorithm that – given developer-provided loop invariants – automatically verifies a program.

- **Case Studies:** I evaluate Leto on several self-correcting algorithms (Jacobi, Self-stabilizing Conjugate Gradient, Self-stabilizing Steepest Descent, and Self-correcting Connected Components) and demonstrate that is possible to verify the key invariants required to prove that these algorithms' self-stability guarantees hold for their implementations.

Leto's contributions enable developers to specify and verify the rich properties seen in applications with application-specific fault tolerance mechanisms.

Chapter 2

Example

Figure 2-1 presents an implementation of the Jacobi iterative method, alternatively Jacobi, in Leto which I will verify against the execution model presented in Figure 1-1. The Jacobi iterative method is an algorithm for solving a system of linear equations. Namely, given a matrix of coefficients `A` and a vector `b` of intercepts, the algorithm computes the solution vector, `x`, where `A * x = b`. The algorithm works iteratively by computing successive approximations of `x`. For a system of two equations (where `A` is a 2x2 matrix and both `b` and `x` are of length two), Jacobi uses the solution vector for the previous iteration, x^k, to produce the solution vector for the current iteration, x^{k+1}, using the following approximation scheme:

$$x_0^{k+1} = (b_0 - A_{0,1} \cdot x_1^k)/A_{0,0}$$
$$x_1^{k+1} = (b_1 - A_{1,0} \cdot x_0^k)/A_{1,1}$$

In words, for a given coordinate x_i, Jacobi approximates x_i^{k+1}, by substituting the values x_j^k, where $i \neq j$, into the linear equation for i, and solving for x_i^{k+1}.

Modulo floating-point rounding error, Jacobi converges to the correct `x` as the number of iterations goes to infinity.

Fault Tolerance. Jacobi is *naturally self-stabilizing*. Namely, given an execution platform in which faults can only affect the calculation of `x` (and do not, for example, corrupt `A`), then Jacobi is in a *valid state* at the end of each iteration: if no additional faults occur during its

```
1  matrix<real>
2  jacobi(int N, matrix<real> A(N, N), matrix<real> b(N),
3         matrix<real> x(N), int iters)
4    requires nzd(A) && 0 < N
5    requires_r eq(N) && eq(A) && eq(b) && eq(x) && eq(iters)
6  {
7    @label(out) for (; 0 <= iters; --iters)
8        invariant_r !model.upset -> eq(x) {
9      matrix<real> next_x(N)
10     specvar int upset_index = 0;
11
12     @label(mid) for (int i = 0; i < N; ++i)
13       invariant 0 <= i < N
14       invariant_r !out[model.upset] -> eq(x)
15       invariant_r (!model.upset ->
16         (!out[model.upset] && eq(next_x)))
17       invariant_r
18         (!out[model.upset] && model.upset) ->
19           bounded_diff_at(next_x,upset_index,i)
20       invariant_r 0 <= upset_index < N<r> {
21
22       real sum = 0;
23       for (int j = 0; j < N; ++j)
24         invariant 0 <= i < N && 0 <= j <= N
25         invariant_r !out[model.upset] -> sig(sum) && eq(j)
26         invariant_r mid[model.upset] -> model.upset {
27
28         if (i != j) {
29           real delta = A[i][j] *. x[j];
30           sum = sum +. delta;
31         }
32       }
33       real num = b[i] - sum;
34       next_x[i] = num / A[i][i];
35
36       if (!mid[model.upset] && model.upset) {
37         upset_index = i;
38       }
39     }
40
41     x = next_x;
42     assert (nzd(A));
43     assert_r (eq(A) &&
44      (!out[model.upset] && model.upset) ->
45        bounded_diff_at(next_x, upset_index, N));
46    }
47
48   return x;
49 }
```

Figure 2-1: Jacobi iterative method

```
1  const real E = ...;
2  const real EPS = ...;
3
4  property nzd(matrix<real> A) :
5    forall(uint fi)((E / EPS) < A[fi][fi] || A[fi][fi] < -(E / EPS));
6
7  property_r sig(real sum) :
8    (!model.upset -> eq(sum)) &&
9    ((!mid[model.upset] && model.upset) ->
10       sum<r> - E < sum<o> < sum<r> + E) &&
11   ((mid[model.upset] && model.upset) -> eq(sum))
12
13 property_r bounded_diff_at(matrix<real> x, int index, int i) :
14   -EPS < x<o>[index] - x<r>[index] < EPS &&
15   forall(fi)((fi < i<r> && (fi != index)) -> x<o>[fi] == x<r>[fi])
```

Figure 2-2: Constants and Properties for Jacobi Iterative Method.

execution, then Jacobi will converge to the correct solution.

To understand this property intuitively, if an iteration produces an incorrect solution vector, then the subsequent execution of the computation is equivalent to having started the computation from scratch with the produced vector as the initial starting point. Moreover, the change in the number of iterations required to converge from the new starting point is bounded logarithmically by the magnitude of the difference in the produced vector from the correct vector.

Verifying Jacobi for a given execution platform therefore poses two challenges:

- Verifying that faults only affect the value of x.
- Identifying a bound on the number of added iterations in the presence of a fault.

Note that the latter determination not only serves as important information for understanding if the implementation will meet the developer's convergence requirements, but it also serves the practical purpose of setting the maximum number of iterations such that a faulty execution will produce a result that is at least as good as a fully reliable execution.

2.1 Jacobi Implementation

The implementation presented in Figure 2-1 stores the matrix of coefficients in the matrix A, the intercepts in the vector b, and the solution vector in x.

The overall architecture of the implementation in Figure 2-1 is that the outer loop on Line 7 computes and stores the solution vector for the current iteration into `next_x`. At the end of each iteration, the implementation updates `x` by copying `next_x` into `x`. The second loop on Line 12 iterates through each x_i (stored at `x[i]`), sums the other terms in the ith equation into the variable `sum` (Line 30), and then computes `x[i]` as the value `(b[i] - sum)/A[i][i]`. I present the definitions and meanings of the properties `nzd`, `sig`, and `bounded_diff_at` in Figure 2-2.

The two features of Leto that diverge from traditional programming languages are that developers can specify that some operations in the program may execute with an alternative semantics and – as consequence – write *relational* assertions that relate values between the standard, *original execution* and the alternative *relaxed execution* of the program. Leto exports customs operations by enabling developers to specify that an operation may execute according to an *execution model specification* by appending a dot to the operation as in the operation `+.` (Line 30). Leto exports relational assertions through `assert_r` statements (Line 44), as well as the ability to specify relational loop invariants with `invariant_r` (Lines 7, 12, and 23) and relational function preconditions with `requires_r` (Line 2).

Execution Model Figure 1-1 presents an *execution model specification* for a *Single Event Upset* (SEU) fault model. The model specification includes the model's *state* and the set of specifications for each operation.

2.2 Specification.

The verified Jacobi implementation relies on Leto's specification capabilities to establish self-stability and verify a convergence bound.

There are two major properties that a developer of Jacobi needs establish:

- Ensure that the resulting implementation is self-stable.
- Determine the worst-case number of iterations added to convergence.

Self-Stability. To verify that this Jacobi implementation is self-stabilizing the developer must verify the property A<o> == A<r>. The notation A<o> refers to the value of A in the standard, original execution of the program whereas the notation A<r> refers to the value of A<r> in the relaxed execution. This property therefore asserts that faults do not disturb the matrix of coefficients. I specify this in the Jacobi implementation within the precondition for the jacobi function using the predicate eq(A), which is shorthand for A<o> == A<r>. The implementation also asserts that this property still holds on each iteration via the assert_r statement on Line 44 and Leto's verification algorithm infers (through a loop invariant inference processes) that that this property holds for all loops within the body of the function (Lines 7, 12, and 23)

Convergence Bound. Jacobi also enjoys a bound on the additional number of iterations added to its execution given a fault. Specifically,

$$\Delta_c = \mathcal{O}\left(\log\left(\frac{1}{\mathsf{EPS}^2}\right)\right)$$

where Δ_c is the number of additional iterations in the relaxed execution and EPS is the maximal impact of error on the x vector. To verify this bound, the developer must verify two properties:

- The perturbation in the solution to due a fault in an iteration is bounded by EPS.
- That $\forall i.\, |A_{i,i}| > \frac{E}{\mathsf{EPS}}$ where E is the maximal error allowed in a single operation from the execution model and EPS is a constant set by the programmer.

The first property bounds the impact of a single fault whereas the second property ensures that errors in the solution vector do not cause the solution vector to experience unbounded increases in the magnitude of its error in subsequent iterations.

I specify this first property with the assert_r on Line 44 which asserts that if

```
!out[model.upset] && model.upset
```

then bounded_diff_at(next_x, upset_index, N). The notation model.upset is a first-

class reference to the state of the execution model at that point in the program. The notation `!out[model.upset]` is a first-class reference to the state of the execution model at the last execution of the label `out`, which corresponds the body of the outer `for` loop (Line 7). This assert statement therefore states that if an upset has occurred during this execution of the loop, then `next_x` differs only at `upset_index` and that difference is bounded. The variable `upset_index` (defined on Line 10) is a *specification variable* that holds state that supports the application's verification but is not reified in the state of the program. I use `upset_index` to specifically record the iteration on which the middle loop encounters a fault (if any). The *specification code* – code used for verification but not execution – one Line 36 appropriately updates `upset_index`'s value.

I specify the second property with the `assert` on Line 42. The predicate `nzd` computes exactly the property that

$$\forall i.\, |A_{i,i}| > \frac{E}{\text{EPS}}.$$

2.3 Verification Approach

I next demonstrate how the developer works together with Leto to verify these assertions. Leto provides an automated verification algorithm that performs *relational* forward symbolic execution to discharge assertions in the program. Namely, Leto traverses the program, building a logical characterization of the state of the program at each point and verifies that the resulting logical formula ensures that a given `assert` or `assert_r` statement is valid. Leto's approach is relational in that it has the ability to track relationships between the standard, *original* execution of the program – for which all operations have their standard semantics – and the *relaxed* execution of the program – for which operations may use their specification as given in the execution model. This approach also works in concert with the developer's specification annotations; these include both function preconditions and loop invariants. Leto also provides support for automatic loop invariant inference, which can lower the annotation burden of the developer by automatically inferring additional loop invariants. Verifying Jacobi with Leto proceeds as follows:

Constants and Properties. Figure 2-2 presents the constants and properties I use in my Jacobi implementation:

- **E.** This constant represents the maximum error experienced by any given operation. It must be less than or equal to E in the execution model (Figure 1-1).
- **EPS.** This constant represents the maximum impact an error may have on any element in the x vector.
- **nzd.** This property takes a matrix A as an input and bounds the absolute value of every element in its diagonal to be greater than E / EPS. This lower bound ensures that EPS bounds the impact of errors on x.
- **sig.** This property takes an input real sum and ensures three invariants:
  - **Line 8.** This conjunct states that sum<o> == sum<r> in the absence of errors.
  - **Line 9.** This conjunct states that if there was an upset while computing sum (!mid[model.upset] && model.upset), then sum<r> is within E of sum<o> (sum<r> - E < sum<o> < sum<r> + E).
  - **Line 11.** This conjunct states that if an upset occurred in a previous iteration (mid[model.upset] && model.upset), then sum<o> == sum<r>.
- **bounded_diff_at.** This property takes input vector x, integer index, and integer i. It asserts that the difference between both executions on x[index] by EPS. It also asserts that x<o> == x<r> at all other indices.

Precondition The verification algorithm starts at the beginning of the jacobi function by assuming that the developer-provided requires and requires_r properties are valid. In this case, the developer states that the input parameters are equal across both executions (Line 5) and that nzd(A).

Outer Loop. Verification next proceeds to the outer loop (Line 7). The outer loop has a single developer-specified invariant: !model.upset -> eq(x). This invariant states that

if no faults have yet to occur, then `x` is equivalent between both the original and relaxed executions. This invariant follows because in the absence of a fault, `jacobi` is a deterministic computation for which any two executions (the original and relaxed execution) that start from the same state (given `jacobi`'s precondition) compute the same result.

Middle Loop. Verification next proceeds to the middle loop on Line 12. This loop uses a *label* to refer to the value of a model state at some control flow point in the program. Namely, the middle loop restates the outer loop's invariant with the notation `!out[model.upset] -> eq(x)`, which states that if the value of `model.upset` at the labeled control flow point `out` is false, then `x` is equivalent. The next invariant refines the outer loop invariant to account for the current state of the execution model. Namely, the invariant `!model.upset -> (!out[model.upset] && eq(next_x))` states that if no upset has occurred, then no upset has occurred previously and in addition, `next_x`, the new value of `x` is equivalent between the original and relaxed execution. This follows because the execution model monotonically sets `model.upset` to `true` and therefore, having seen no fault, the computations executes correctly and deterministically to produces `next_x`. The next invariant is the crux of the verification approach in that it connects the behavior of the computation without observing a fault to that after observing the fault. Namely, the invariant

```
!out[model.upset] && model.upset -> bounded_diff_at(next_x, upset_index, i)
```

states that if no fault has occurred previously, but a fault has occurred on this iteration, then `next_x` differs in at most one position and the difference in that position is bounded. This follows because of the single-event upset nature of the execution model and the fact that errors in the result of an operation are appropriately bounded.

The invariant (`0 <= upset_index < N<o>`) serves to pass information about the bound on `upset_index` to the outer loop. Using this invariant, Leto knows that the vector access in `bounded_diff_at` (Line 45) is in bounds.

Inner Loop. All of the relaxation in my Jacobi implementation comes from this loop, where it may corrupt the value of sum. The invariant (!out[model.upset] -> sig(sum)) verifies that if an error occurred on this inner loop iteration, then sum<r> is within E of sum<o>. Otherwise, sum<r> == sum<o>. The bound in this invariant passes this error information on to the outer loops so that they may reason about the impact of a corrupted sum. The second invariant (mid[model.upset] -> model.upset) asserts that if an upset had previously occurred, then the upset variable in the model is still *true*.

The invariant eq(j) ensures that the inner loop executes in lockstep. The other two invariants (0 <= i < N and 0 <= j <= N) ensure that accesses to elements in A and x (Line 29) are in bounds.

2.4 Verification Algorithm

For Jacobi, the algorithm performs the following steps:

Preprocessing. A preprocessing step converts the program to static single assignment form. For each variable declaration x, Leto constructs two variables named x<o> and x<r> representing the variables in the original and relaxed executions respectively. Specification variables only exist in the relaxed context so the declaration of upset_index, for example, will result in the construction of upset_index and not upset_index<r> as the <r> suffix is superfluous.

Relaxed Binary Operators. When encountering an expression of the form $e_1 \oplus. e_2$, Leto examines the execution model to identify the specified operators matching $\oplus$ and emits the constraint $\bigvee_{\oplus}(b_1 \rightarrow b_2)$ where b_1 is the content of the **when** clause for $\oplus$ and b_2 is the **ensures** clause. This constraint captures the property that the result of the binary operation may be computed by any of the relaxed operations where b_1 is satisfied.

Loops. When reaching a loop, Leto first verifies that the loop invariant holds. After validating the loop invariant, Leto determines which execution paths are possible. Namely,

there are three possibilities for how two executions of the program may execute through the loop:

- In *lockstep* with both executions either executing the body of the loop or terminating.
- *Desynchronized* with the original, reliable execution executing the body of the loop and the relaxed execution terminating.
- Desynchronized, with the reliable execution terminating and the relaxed execution executing the body of the loop. Leto determines which paths are possible by checking the satisfiability of the loop condition under a context containing only the loop invariant and then recursively verifies the possible executions.

After the body of the loop is checked, Leto verifies that the loop invariant holds. If so, the loop invariant and $(\neg b_o \wedge \neg b_r)$ are added to the constraint list that existed prior to entering the loop.

In the case of Jacobi, all loops are checked in lockstep as the invariants for each constrain the various iteration counters and bounds to be equal across both executions.

Additionally, Leto uses inference to infer invariants of the form `eq(x)` for all variables `x`. It will also try to lift invariants from the immediate parent loop or function. I detail the inference algorithm in Section 5.1.

Conditionals. Like `while` loops, `if` statements have multiple paths to consider:

- Both executions execute the *true* branch.
- Both executions execute the *false* branch.
- The original execution executes the *true* branch and the relaxed execution executes the *false* branch.
- The original execution executes the *false* branch and the relaxed executes the *true* branch.

As with `while` loops, Leto recursively verifies each (satisfiable) possibility.

Assertions Jacobi contains an assert_r on Line 44. When encountering a assert_r, Leto first checks that the conjunction of the constraint list and the negation of the assert condition is false. If this is not the case, Leto reports the assertion failure and exits. If the assertion holds, it is added to the list of constraints.

Jacobi contains an assert on Line 42. In lockstep execution assert(b) is equivalent to assert_r(b<o> -> b<r>). If original program is executing out of lockstep, then Leto adds b<o> to its context. If the relaxed program is executing out of lockstep, then assert(b) is equivalent to assert_r(b<r>).

assume(b) has the same semantics as assert(b) in Leto to verify that faults do not interfere with the reasoning behind assumptions.

Chapter 3

Language

Figure 3-1 presents Leto's programming language. Leto provides a general-purpose imperative language that includes specification primitives (e.g., `requires`) in the spirit of ESC/Java [17] to support verifying applications.

Functions and Properties. A program π consists of a sequence of function and property declarations. A function declaration F specifies the function's return type, id, parameters, preconditions and the code of the function, as a statement s. A property declaration P defines a hygenic macro and my be either unary or relational. A unary property declaration (uses keyword `property`) consists of an id, a list of parameters, and a unary boolean expression. A relational property declaration (uses keyword `property_r`) constists of an id, a list of parameters, and a relational boolean expression.

Data Types. The language includes primitive data types (τ) of (signed and unsigned) integers, reals, and booleans as well as vectors/matrices of these types. A developer can use the `@region`(r) annotation to state the variable is allocated in a named memory region, r, for which reads as writes may have a custom semantics according to the execution model.

Expressions. Leto includes standard numerical operations, comparison, and logical expressions, along with dotted notations (e.g., `x +. y`) that communicate that the operation may have a custom semantics as specified in the execution model.

$c \in$ constants, $x \in$ variables, $r \in$ memory regions
$l \in$ labels, $f \in$ functions, $p \in$ properties
$p_r \in$ relational properties

$$\pi \rightarrow P \mid F \mid \pi \;;\; \pi$$
$$F \rightarrow T\ f\ (D^*)\ \langle \texttt{requires}\ B_v\rangle^*\ \langle \texttt{requires_r}\ B_r\rangle^*\ \{\ S\ \}$$
$$P \rightarrow \texttt{property}\ p\ (\langle T\ x\rangle^*)\ :\ B_v$$
$$\mid \texttt{property_r}\ p\ (\langle T\ x\rangle^*)\ :\ B_r$$
$$S \rightarrow D \mid x = E \mid S\ ;\ S \mid \texttt{if}\ (B)\ \{\ S\ \}\ \texttt{else}\ \{\ S\ \}$$
$$\mid \langle \texttt{@label}\ l\rangle^*\ \texttt{while}\ (B)\ I^*\ \{\ S\ \} \mid \texttt{return}\ E$$
$$\mid \langle \texttt{@label}\ l\rangle^*\ \texttt{for}\ (S\ ;\ B\ ;\ S)\ I^*\ \{\ S\ \} \mid \texttt{skip}$$
$$\mid \texttt{assume}\ (B_v) \mid \texttt{assert_r}\ (B_r) \mid \texttt{assert}\ (B_v)$$

$$I \rightarrow \texttt{invariant}\ B_v \mid \texttt{invariant_r}\ B_r$$
$$D \rightarrow \langle \texttt{@region}\ r\rangle^?\ \tau\ x \mid \langle \texttt{@region}\ r\rangle^?\ \texttt{matrix<}\tau\texttt{>}\ x\ c$$
$$\tau \rightarrow \texttt{int} \mid \texttt{uint} \mid \texttt{real} \mid \texttt{bool}$$
$$T \rightarrow \tau \mid \texttt{matrix <}\tau\texttt{>}$$

$$M \rightarrow \texttt{model}\ \{\ D^*\ O^+\ \}$$
$$O \rightarrow \texttt{operator}\ Op\ (x_s^+)\ C^+$$
$$Op \rightarrow \neg \mid \neg. \mid \oplus \mid \prec \mid \diamond$$
$$\mid \texttt{@region}\ (r)\ \texttt{read} \mid \texttt{@region}\ (r)\ \texttt{write}$$
$$C \rightarrow \texttt{when}\ (B_r) \mid \texttt{ensures}\ (B_r)$$
$$\mid \texttt{modifies}\ (x_s^+)$$

$$\diamond \rightarrow \wedge \mid \wedge. \mid \vee \mid \vee. \mid \rightarrow \mid \rightarrow.$$
$$\prec \rightarrow < \mid <. \mid \leq \mid \leq. \mid = \mid =. \mid \neq \mid \neq.$$
$$\oplus \rightarrow + \mid +. \mid - \mid -. \mid \times \mid \times. \mid \div \mid \div.$$

$$E \rightarrow c \mid E\ \oplus E \mid \texttt{model.}x \mid x \mid x[E]$$
$$E_r \rightarrow Q \mid E_r\ \oplus E_r \mid Q[E_r] \mid E$$
$$Q \rightarrow x\texttt{<o>} \mid x\texttt{<r>} \mid x$$

$$B \rightarrow \mathit{true} \mid \mathit{false} \mid E\ \prec\ E \mid B\ \diamond\ B$$
$$\mid f\ (E^*) \mid \neg\ B \mid \neg.\ B$$
$$B_v \rightarrow \forall\ x_s\ B \mid \exists\ x_s\ B \mid p\ x^* \mid B$$
$$B_r \rightarrow \mathit{true} \mid \mathit{false} \mid E_r\ \prec\ E_r \mid B_r\ \diamond\ B_r$$
$$\mid \neg\ B_r \mid \neg.\ B_r \mid \forall\ x\ B_r \mid \exists\ x\ B_r$$

Figure 3-1: Language Syntax

Memory Operations. Leto supports reads from and writes to variables, including values of both primitive and array/matrix type. Reads and writes to variables allocated in a designated memory region will operate with the semantics as given in the program's execution model.

Assertions and Assumptions. Leto also enables a developer to specify assertions and assumptions on the state of the program. Leto's language includes both standard `assert` statements and `assume` statements (with their traditional meaning). Each such statement can use a quantified boolean expression, B_v, that quantifies over the value of variable (e.g., the index of an array/matrix). A relational assertion statement, `assert_r`, uses a quantified relational boolean expression, B_r, that specifies a relationship between the original and relaxed executions to verify.

Control flow. Leto's language includes standard control constructs, such as sequential composition, `if` statements, `while` loops, and `for` loops. A developer can specify a named *label* for a `while` or `for` loop via the `@label`(l) annotation. As in Jacobi, such labels enable the developer to refer to the model state at a specific point in the execution of the program. For `while` and `for` statements, a developer can additionally specify *loop invariants* to support verification via the syntax `invariant` (unary loop invariant) and `invariant_r` (relational loop invariant). A loop invariant specifies a property that must be true on entry to the loop, as well as at the end of each loop iteration. Loop invariants are a key to verifying applications that contain loops because automatically inferring loop invariants is undecidable in general. Therefore, a developer may need to specify additional loop invariants when Leto's loop invariant inference procedure is insufficient.

Execution Model. An execution model M consists of a set of state variables x^* and *operation specifications* (O^*). Each operation specification (O) specifies:

- The target operator for the specification.
- A list of variables as parameters to the specification.
- A set of clauses.

A clause is either a `when` clause, which guards the execution of the specification with a predicate P, an `ensures` clause, which establishes a relationship on the output of the specification given the inputs to the specification and execution model's state variables, or a `modifies` clause, that specifies which of the model's state variables changes as a result of using the operation. Predicates consist of standard operations over standard expressions with the addition of the distinguished `result` variable, which captures the result of the specification's execution.

3.1 Dynamic Semantics

Figure 3-2 presents an abbreviated dynamic semantics of Leto's language. I present an extended semantics in Appendix A. Leto's semantics models an abstract machine that includes a *frame*, a *heap*, and an *execution model state.* Leto allocates memory for program variables (both scalar and array) in the heap. A frame serves two roles:

- A frame maps a program variable to the address of the memory region allocated for that variable in the heap.
- A frame maps a register to its current value in the program.

The model state stores the values for state variables within the execution model.

3.1.1 Preliminaries

Frames, Heaps, Model States, and Environments. A *frame*

$$\sigma \in \Sigma = \mathrm{Var} \cup \mathrm{Reg} \rightarrow \mathrm{Int}_{\mathrm{N}}$$

is a finite map from variables and registers to N-bit integers. A *heap*

$$h \in H = \mathrm{Loc} \rightarrow \mathrm{Int}_{\mathrm{N}}$$

F-BINOP

$$\frac{\mu(\oplus, [x_1, x_2], P_w, P_e) \quad m[x_1 \mapsto n_1][x_2 \mapsto n_2] \models P_w \qquad m'[x_1 \mapsto n_1][x_2 \mapsto n_2][\mathit{result} \mapsto n_3] \models P_e \qquad \mathrm{dom}(m) = \mathrm{dom}(m')}{\langle \mathtt{model.} \oplus (n_1, n_2),\ m\rangle \Downarrow_\mu \langle n_3,\ m'\rangle}$$

BINOP

$$\frac{n_1 = \sigma(r_1) \qquad n_1 = \sigma(r_2) \qquad \langle m, \oplus, (n_1, n_2)\rangle \Downarrow_\mu \langle n_3,\ m'\rangle}{\langle r = r_1 \oplus r_2, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \xrightarrow{\mu} \langle r = n_3, \langle \sigma,\ h,\ \theta,\ m'\rangle\rangle}$$

READ

$$\frac{a = \sigma(x) \qquad n = h(a) \qquad q = \theta(a) \qquad \langle m, \mathtt{read}, (n, q)\rangle \Downarrow_\mu \langle n',\ m'\rangle}{\langle r = x, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \xrightarrow{\mu} \langle r = n', \langle \sigma,\ h,\ \theta,\ m'\rangle\rangle}$$

WRITE

$$\frac{a = \sigma(x) \quad n_{old} = h(a) \qquad n_{new} = \sigma(r) \qquad q = \theta(a) \qquad \langle m, \mathtt{write}, (n_{old}, n_{new}, q)\rangle \Downarrow_\mu \langle n_r,\ m'\rangle}{\langle x = r, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \xrightarrow{\mu} \langle \mathtt{skip}, \langle \sigma,\ h[a \mapsto n_r],\ \theta,\ m'\rangle\rangle}$$

ASSIGN

$$\frac{}{\langle r = n, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \xrightarrow{\mu} \langle \mathtt{skip}, \langle \sigma[r \mapsto n],\ h,\ \theta,\ m\rangle\rangle}$$

ASSERT-T

$$\frac{\pi_1(\varepsilon)(r) = \mathit{true}}{\langle \mathtt{assert}\ r, \varepsilon\rangle \xrightarrow{\mu} \langle \mathtt{skip}, \varepsilon\rangle}$$

ASSERT-F

$$\frac{\pi_1(\varepsilon)(r) = \mathit{false}}{\langle \mathtt{assert}\ r, \varepsilon\rangle \xrightarrow{\mu} \langle \mathtt{fail}, \varepsilon\rangle}$$

ASSUME

$$\frac{}{\langle \mathtt{assume}\ r, \varepsilon\rangle \xrightarrow{\mu} \langle \mathtt{assert}\ r, \varepsilon\rangle}$$

IF-T

$$\frac{\pi_1(\varepsilon)(r) = \mathit{true}}{\langle \mathtt{if}\ r\ s_1\ s_2, \varepsilon\rangle \xrightarrow{\mu} \langle s_1, \varepsilon\rangle}$$

IF-F

$$\frac{\pi_1(\varepsilon)(r) = \mathit{false}}{\langle \mathtt{if}\ r\ s_1\ s_2, \varepsilon\rangle \xrightarrow{\mu} \langle s_2, \varepsilon\rangle}$$

SEQ1

$$\frac{\langle s_1, \varepsilon\rangle \xrightarrow{\mu} \langle s_1', \varepsilon'\rangle}{\langle s_1\ ; s_2, \varepsilon\rangle \xrightarrow{\mu} \langle s_1'\ ; s_2,\ \varepsilon'\rangle}$$

SEQ2

$$\frac{}{\langle \mathtt{skip}\ ;\ s_2, \varepsilon\rangle \xrightarrow{\mu} \langle s_2, \varepsilon\rangle}$$

WHILE-F

$$\frac{\pi_1(\varepsilon)(r) = \mathit{false}}{\langle \mathtt{while}\ r\ s, \varepsilon\rangle \xrightarrow{\mu} \langle \mathtt{skip}, \varepsilon\rangle}$$

WHILE-T

$$\frac{\pi_1(\varepsilon)(r) = \mathit{true}}{\langle \mathtt{while}\ r\ s, \varepsilon\rangle \xrightarrow{\mu} \langle s\ ;\ \mathtt{while}\ r\ s, \varepsilon\rangle}$$

Figure 3-2: Semantics of Execution Model and Instructions

is a finite map from locations ($n \in \text{Loc} \subset \text{Int}_\text{N}$) to N-bit integer values. A *region map*

$$\theta \in \Theta = \text{Loc} \rightarrow \text{Region}$$

is a finite map from locations to memory regions. A *model state*

$$m \in M = \text{Var} \rightarrow \text{Int}_\text{N}$$

is a finite map from model state variables to N-bit integer values. An *environment*

$$\varepsilon \in \text{E} = \Sigma \times H \times M$$

is a tuple consisting of a frame, a heap, and a model state. An *execution model specification*

$$\mu \subseteq Op \times list(\text{Var}) \times P \times P$$

is a relation consisting of tuples of an operation $op \in Op$, a list of variables, and two unary logical predicates.

Initialization. For clarity of presentation, I assume a compilation and execution model in which memory locations for program variables are allocated and the corresponding mapping in the frame are done prior to execution of the program (similar in form to c-style declarations).

3.1.2 Execution Model Semantics

Figure 3-2 provides an abbreviated presentation of the execution model relation

$$\langle \texttt{model}.op(args),\ m\rangle \Downarrow_\mu \langle n,\ m'\rangle.$$

The relation states that given the arguments, *args* to an operation *op*, evaluation of the operation from the model state m yields a result n and a new model state m' under the execution model specification μ.

The [F-BINOP] rule specifies the meaning of this relation for binary operations. This relation

states that the value of an operation $\oplus(n_1, n_2)$ given an execution model state m evaluates to value n_3 and a new model state m'. The rule relies on the relation $\mu(op, vlist, P_w, P_e)$ which specifies the list of argument names, *vlist*, the *precondition* P_w and the *postcondition* P_e for the operation *op* in the developer-provided execution model. The precondition of an operation is the conjunction of the `when` clauses in the operation's specification. The postcondition of an operation is the conjunction of the `ensures` clauses in the operation's specification.

The semantics of the model relation non-deterministically selects an operation specification, result value, and output model state subject to the constraint that:

- The current model state satisfies the precondition (after the inputs to the operation have been appropriately assigned into the model state).
- The output model state satisfies the postcondition (after the inputs and result value have been appropriated assigned into the model state).
- The domains of the input and output state are the same.

Because of the uniformity of the execution model specification, the semantics for other operations in the program is similar with the sole distinction being the number of arguments passed to the operation. For clarity of presentation, I elide those rules.

3.1.3 Language Semantics

Figure 3-2 presents the non-deterministic small-step transition relation $\langle s, \varepsilon \rangle \xrightarrow{\mu} \langle s', \varepsilon' \rangle$ of a Leto program. The relation states that execution of statement s from the environment ε takes one step yielding the statement s' and environment ε' under the execution model specification μ. The semantics of the statements are largely similar to that of traditional approaches except for the ability to the statements to encounter faults. Broadly, I categorize Leto's instructions into four categories: *register instructions*, *memory instructions*, *assertions*, and *control flow*.

Register Instructions. The rules [ASSIGN] and [BINOP] specify the semantics of two of Leto's register manipulation instructions. [ASSIGN] defines the semantics of assigning an

integer value to a register, $r = n$. This has the expected semantics updating the value of r within the current frame with the value n. Of note is that register assignment executes fully reliably without faults.

[BINOP] specifies the semantics of a register only binary operation, $r = r_1 \oplus r_2$. Note that reads of the input registers execute fully reliably. The result of the operation is n_3, which is the value of the operation given the semantics of that operation's execution model when executed from the model state m on parameters n_1 and n_2. Executing the execution model may change the values of the execution model's state variables. Therefore, the instruction evaluates to an instruction that assigns n_3 to the destination register and evaluates with a environment that consists of the unmodified frame, the unmodified heap, and the modified execution model state. Note that by virtue of the fact that both the frame and heap are unmodified, faults in register instructions cannot modify the contents or organization of memory. This modeling choice is consistent with standard fault modeling approaches.

Memory Instructions. The rules [READ] and [WRITE] specify the semantics of two of Leto's memory manipulation instructions. [READ] defines the semantics of reading the value of a program variable x from it's corresponding memory location: $r = x$. The rule fetches the program variable's memory address from the frame, reads the value of the memory location $n = h(a)$ and the region the memory location belongs to $q = \theta(a)$ and then executes the execution model with the program variable's current value in memory and the memory region it resides in as a parameters. The execution model non-deterministically yields a result n' that the rule uses to complete its implementing by issuing an assignment to the register.

[WRITE] defines the semantics of writing the value of a register to memory. The rule reads the value of the memory location to record the old value of the memory location, reads the value of the input register, fetches the region the memory location corresponds to, and then executes the execution model with these values as parameters. The execution model yields a new value n_r that the rule then assigns to the value the program variable.

Assertions. The rules [ASSERT-T] and [ASSUME-T] specify the semantics of `assert` and `assume` statements, respectively. These statements have standard semantics, yielding a `skip`

and continuing the execution of the program if their conditions are satisfied. For either of these statements, if their conditions evaluate to false, then execution yields `fail` denoting that execution has failed and become stuck in error.

Control Flow. The rules for control flow ([IF-T], [IF-F], [SEQ1], [SEQ2], [WHILE-F], and [WHILE-T]) have standard semantics. An important note is that the semantics of these statements is such that the transfer of control from one instruction to another always executes reliably and, therefore, faults do not introduce control flow errors into the program. This modeling assumption is consistent with standard fault injection and reliability analysis models [65].

Chapter 4

Logic

I next present the program logic for verifying relaxed executions of a Leto program. The program logic is a relational program logic in that it relates relaxed executions of the program to its original, reliable execution. Before delving into the details of the logic, I first present preliminary definitions, including an explanation of the underlying relational assertion logic.

4.1 Preliminaries

Assertion Logic Syntax. Figure 4-1 presents the syntax of my relational assertion logic. Note that the presentation includes both standard expressions and predicates (E and P), respectively, as well as relational expressions RE and predicates RP. Standard expressions can reference integer values, registers, and program variables. Standard expressions also include standard integer operations. Relational predicates build on top of standard predicates, enabling the specification of comparisons between expressions as well as logical combinations of properties.

Relational expressions extend standard expressions with the ability to refer to values of program quantities in the standard execution of the program as well as in the relaxed execution. For example, via the syntax r`<o>` and r`<r>`, the logic can distinguish between the value of the registers r between both executions. An additional quantity in relational expressions is the ability to refer to the execution model state. Specifically, the syntactic construct `model.x` enables a developer to refer to the value of variables in the execution

$$\texttt{iop} ::= + \mid - \mid * \mid / \mid \ldots$$
$$\texttt{cmpop} ::= < \mid > \mid = \mid \ldots$$
$$\texttt{lop} ::= \wedge \mid \vee \mid \ldots$$

$$E ::= \texttt{n} \mid \texttt{r} \mid \texttt{x} \mid E \ \texttt{iop} \ E$$
$$RE ::= \texttt{n} \mid \texttt{r<o>} \mid \texttt{r<r>} \mid \texttt{x<o>} \mid \texttt{x<r>} \mid \texttt{model.x} \mid RE \ \texttt{iop} \ RE$$
$$P ::= \texttt{true} \mid \texttt{false} \mid E \ \texttt{cmpop} \ E \mid P \ \texttt{lop} \ P \mid \neg P$$
$$RP ::= \texttt{true} \mid \texttt{false} \mid RE \ \texttt{cmpop} \ RE \mid RP \ \texttt{lop} \ RP \mid \neg RP$$

$$\begin{aligned}
[\![E]\!] &\in \mathrm{E} \rightarrow \mathbb{Z} \\
[\![\texttt{n}]\!](\varepsilon) &= n \\
[\![\texttt{r}]\!](\varepsilon) &= \pi_1(\varepsilon)(x) \\
[\![\texttt{x}]\!](\varepsilon) &= \pi_2(\varepsilon)(\pi_1(\varepsilon)(x)) \\
[\![\ E_1 \ \texttt{iop} \ E_2]\!](\varepsilon) &= [\![E_1]\!](\varepsilon) \ iop \ [\![E_2]\!](\varepsilon)
\end{aligned}$$

$$\begin{aligned}
[\![RE]\!] &\in \mathrm{E} \times \mathrm{E} \rightarrow \mathbb{Z} \\
[\![\texttt{n}]\!](\varepsilon_1, \varepsilon_2) &= n \\
[\![\texttt{r<o>}]\!](\varepsilon_1, \varepsilon_2) &= \pi_1(\varepsilon_1)(r) \\
[\![\texttt{r<r>}]\!](\varepsilon_1, \varepsilon_2) &= \pi_1(\varepsilon_2)(r) \\
[\![\texttt{x<o>}]\!](\varepsilon_1, \varepsilon_2) &= \pi_2(\varepsilon_1)(\pi_1(\varepsilon_1)(x)) \\
[\![\texttt{x<r>}]\!](\varepsilon_1, \varepsilon_2) &= \pi_2(\varepsilon_2)(\pi_1(\varepsilon_2)(x)) \\
[\![RE_1 \ \texttt{iop} \ RE_2]\!](\varepsilon_1, \varepsilon_2) &= [\![RE_1]\!](\varepsilon_1, \varepsilon_2) \ iop \ [\![RE_2]\!](\varepsilon_1, \varepsilon_2) \\
[\![\texttt{model.x}]\!](\varepsilon_1, \varepsilon_2) &= \pi_3(\varepsilon_2)(x)
\end{aligned}$$

$$\begin{aligned}
[\![P]\!] &\in \mathcal{P}(\mathrm{E}) \\
[\![\texttt{true}]\!] &= \mathrm{E} \quad [\![\texttt{false}]\!] = \emptyset \\
[\![E_1 \ \texttt{cmpop} \ E_2]\!] &= \{\varepsilon \mid [\![E_1]\!](\varepsilon) \ cmp \ [\![E_2]\!](\varepsilon)\} \\
[\![P_1 \ \texttt{lop} \ P_2\ ]\!] &= \{\varepsilon \mid \varepsilon \in [\![P_1]\!] \ lop \ \varepsilon \in [\![P_2]\!]\} \\
[\![\neg P]\!] &= [\![\texttt{true}]\!] \setminus [\![P]\!] \\
[\![RP]\!] &\in \mathcal{P}(\mathrm{E} \times \mathrm{E}) \\
[\![\texttt{true}]\!] &= \mathrm{E} \times \mathrm{E} \quad [\![\texttt{false}]\!] = \emptyset \\
[\![RE_1 \ \texttt{cmpop} \ RE_2]\!] &= \{(\varepsilon_1, \varepsilon_2) \mid [\![RE_1]\!](\varepsilon_1, \varepsilon_2) \ cmp \ [\![RE_2]\!](\varepsilon_1, \varepsilon_2)\} \\
[\![RP_1 \ \texttt{lop} \ RP_2]\!] &= \{(\varepsilon_1, \varepsilon_2) \mid (\varepsilon_1, \varepsilon_2) \in [\![RP_1]\!] \ lop \ (\varepsilon_1, \varepsilon_2) \in [\![RP_2]\!]\} \\
[\![\neg RP]\!] &= [\![\texttt{true}]\!] \setminus [\![RP]\!]
\end{aligned}$$

Figure 4-1: Relational Assertion Logic Syntax and Semantics

model's state in the relaxed execution. Relational predicates (RP) build upon relational expressions to enable comparison between relational expressions and logical combinations of relational predicates.

Assertion Logic Semantics. Figure 4-1 presents the denotational semantics of my relational assertion logic. I model the semantics of a predicate as the set of environments that satisfy the predicate. The semantic function $[\![P]\!]$ gives the semantics to standard predicates. I model the semantics of a relational predicate as the set of environment pairs – one environment for the standard execution and one for the relaxed execution – that satisfy the predicate. The semantic function $[\![RP]\!]$ gives the semantics to relational predicates. Both semantic functions build upon the semantics of standard and relational expressions, respectively. The semantic function $[\![E]\!]$ provides the semantics for standard expressions whereas $[\![RE]\!]$ provides the semantics for relational expressions. I model these semantics, respectively, as a function from an environment to an N-bit integer and a function from an environment pair to an N-bit integer.

Auxiliary Definitions. To support the formalization in the remainder of the paper I define the following auxiliary notation:

$$
\begin{array}{rlrl}
inj_t(\texttt{n}) &= n & inj_t(\texttt{true}) &= \texttt{true} \\
inj_t(\texttt{r}) &= \texttt{r<t>} & inj_t(\texttt{false}) &= \texttt{false} \\
inj_t(\texttt{x}) &= \texttt{x<t>} & inj_t(E_1 \texttt{ cmpop } E_1) &= inj_t(E_1) \texttt{ cmpop } inj_t(E_2) \\
inj_t(E_1 \texttt{ iop } E_1) &= inj_t(E_1) \texttt{ iop } inj_t(E_2) & inj_t(P_1 \texttt{ lop } P_1) &= inj_t(P_1) \texttt{ lop } inj_t(P_2) \\
 & & inj_t(\neg P) &= \neg inj_t(P)
\end{array}
$$

The function $inj_t(\cdot)$ where $t \in \{o, r\}$ implements an *injection* for standard unary predicates into a relational domain. For $t = o$, the definition injects a predicate into the domain of the reliable execution of the program whereas when $t = r$, the definition injects a predicate into the domain of the relaxed execution.

ASSIGN-L

$$\frac{}{\vdash_l \{\ RQ[n/inj_o(r)]\ \}\ r = n\ \{\ RQ\ \}}$$

BINOP-L

$$\frac{}{\vdash_l \{\ RQ[inj_o(r_1 \oplus r_2)/inj_o(r)]\ \}\ r = r_1 \oplus r_2\ \{\ RQ\ \}}$$

READ-L

$$\frac{}{\vdash_l \{\ RQ[inj_o(x)/inj_o(r)]\ \}\ r = x\ \{\ RQ\ \}}$$

WRITE-L

$$\frac{}{\vdash_l \{\ RQ[inj_o(r)/inj_o(x)]\ \}\ x = r\ \{\ RQ\ \}}$$

ASSERT-L

$$\frac{}{\vdash_l \{\ true\ \}\ \texttt{assert}\ r\ \{\ inj_o(r)\ \}}$$

ASSUME-L

$$\frac{\vdash_l \{\ true\ \}\ \texttt{assert}\ r\ \{\ RQ\ \}}{\vdash_l \{\ true\ \}\ \texttt{assume}\ r\ \{\ RQ\ \}}$$

RELATIONAL-ASSERT-L

$$\frac{}{\vdash_l \{\ true\ \}\ \texttt{assert_r}\ r\ \{\ true\ \}}$$

ASSIGN-R

$$\frac{}{\mu \vdash_r \{\ RQ[n/inj_r(r)]\ \}\ r = n\ \{\ RQ\ \}}$$

BINOP-R

$$\frac{\text{fresh}(r') \qquad RQ' = RQ[inj_r(r')/inj_r(r)] \wedge \big(\forall([x_1, x_2], P_w, P_e) \in \mu(\oplus) \cdot inj_r(P_w[r_1/x_1][r_2/x_2]) \rightarrow inj_r(P_e[r'/result])\big)}{\mu \vdash_r \{\ RQ'\ \}\ r = r_1 \oplus r_2\ \{\ RQ\ \}}$$

READ-R

$$\frac{\text{fresh}(r') \qquad RQ' \equiv RQ[inj_r(r')/inj_r(r)] \wedge \big(\forall([x_{mem}], P_w, P_e) \in \mu(\texttt{read}) \cdot inj_r(P_w[x/x_{mem}]) \rightarrow inj_r(P_e[r'/result])\big)}{\mu \vdash_r \{\ RQ'\ \}\ r = x\ \{\ RQ\ \}}$$

WRITE-R

$$\frac{\text{fresh}(r') \qquad RQ' = RQ[inj_r(r')/inj_r(x)] \wedge \big(\forall([x_1, x_2], P_w, P_e) \in \mu(\texttt{write}) \cdot inj_r(P_w[x/x_1][r/x_2]) \rightarrow inj_r(P_e[r'/x_1])\big)}{\mu \vdash_r \{\ RQ'\ \}\ x = r\ \{\ RQ\ \}}$$

ASSERT-R

$$\frac{}{\mu \vdash_r \{\ inj_r(r)\ \}\ \texttt{assert}\ r\ \{\ inj_r(r)\ \}}$$

ASSUME-R

$$\frac{\mu \vdash_r \{\ inj_r(r)\ \}\ \texttt{assert}\ r\ \{\ inj_r(r)\ \}}{\mu \vdash_r \{\ inj_r(r)\ \}\ \texttt{assume}\ r\ \{\ inj_r(r)\ \}}$$

RELATIONAL-ASSERT-R

$$\frac{}{\mu \vdash_r \{\ r\ \}\ \texttt{assert_r}\ r\ \{\ r\ \}}$$

Figure 4-2: Left and Right Rules for Primitive Statements

SEQ-L
$$\frac{\vdash_l \{\ RP\ \}\ s_1\ \{\ RR\ \} \qquad \vdash_l \{\ RR\ \}\ s_2\ \{\ RQ\ \}}{\vdash_l \{\ RP\ \}\ s_1\ ;\ s_2\ \{\ RQ\ \}}$$

WHILE-L
$$\frac{b \equiv inj_o(r) = true}{\vdash_l \{\ true\ \}\ \texttt{while}\ r\ RP\ s\ \{\ RP \wedge \neg b\ \}}$$

IF-L
$$\frac{b \equiv inj_o(r) = true \qquad \vdash_l \{\ RP \wedge b\ \}\ s\ \{\ RQ\ \} \qquad \vdash_l \{\ RP \wedge \neg b\ \}\ s\ \{\ RQ\ \}}{\vdash_l \{\ RP\ \}\ \texttt{if}\ r\ s_1\ s_2\ \{\ RQ\ \}}$$

SEQ-R
$$\frac{\mu \vdash_r \{\ RP\ \}\ s_1\ \{\ RR\ \} \qquad \mu \vdash_r \{\ RR\ \}\ s_2\ \{\ RQ\ \}}{\mu \vdash_r \{\ RP\ \}\ s_1\ ;\ s_2\ \{\ RQ\ \}}$$

WHILE-R
$$\frac{b \equiv inj_o(r) = true \qquad \mu \vdash_r \{\ RP \wedge b\ \}\ s\ \{\ RP\ \}}{\mu \vdash_r \{\ RP\ \}\ \texttt{while}\ r\ RP\ s\ \{\ RP \wedge \neg b\ \}}$$

IF-R
$$\frac{b \equiv inj_o(r) = true \qquad \mu \vdash_r \{\ RP \wedge b\ \}\ s\ \{\ RQ\ \} \qquad \mu \vdash_r \{\ RP \wedge \neg b\ \}\ s\ \{\ RQ\ \}}{\mu \vdash_r \{\ RP\ \}\ \texttt{if}\ r\ s_1\ s_2\ \{\ RQ\ \}}$$

Figure 4-3: Left and Right Rules for Structure and Control Flow

STAGE
$$\frac{\vdash_l \{\ RP\ \}\ s_1\ \{\ RR\ \} \qquad \mu \vdash_r \{\ RR\ \}\ s_2\ \{\ RQ\ \}}{\mu \vdash \{\ RP\ \}\ s_1 \sim s_2\ \{\ RQ\ \}}$$

SPLIT
$$\frac{\mu \vdash \{\ RP\ \}\ s \sim s\ \{\ RQ\ \}}{\mu \vdash \{\ RP\ \}\ s\ \{\ RQ\ \}}$$

SEQ
$$\frac{\mu \vdash \{\ P\ \}\ s_1\ \{\ R\ \} \qquad \mu \vdash \{\ R\ \}\ s_2\ \{\ Q\ \}}{\mu \vdash \{\ P\ \}\ s_1\ ;\ s_2\ \{\ Q\ \}}$$

WEAK
$$\frac{RP \models RP' \qquad \mu \vdash \{\ RP'\ \}\ s\ \{\ RQ'\ \} \qquad RQ' \models RQ}{\mu \vdash \{\ RP\ \}\ s\ \{\ RQ\ \}}$$

IF
$$\frac{\begin{array}{c} b \equiv r = true \\ \mu \vdash \{\ P \wedge inj_o(b) \wedge inj_r(b)\ \}\ s_1\ \{\ Q\ \} \qquad \mu \vdash \{\ P \wedge \neg inj_o(b) \wedge inj_r(b)\ \}\ s_2 \sim s_1\ \{\ Q\ \} \\ \mu \vdash \{\ P \wedge inj_o(b) \wedge \neg inj_r(b)\ \}\ s_1 \sim s_2\ \{\ Q\ \} \qquad \mu \vdash \{\ P \wedge \neg inj_o(b) \wedge \neg inj_r(b)\ \}\ s_2\ \{\ Q\ \} \end{array}}{\vdash \{\ P\ \}\ \texttt{if}\ r\ s_1\ s_2\ \{\ Q\ \}}$$

WHILE
$$\frac{\begin{array}{c} b \equiv r = true \qquad \vdash \{\ R \wedge inj_o(b) \wedge inj_r(b)\ \}\ s\ \{\ R\ \} \\ \vdash_l \{\ R \wedge inj_o(b) \wedge \neg inj_r(b)\ \}\ s\ \{\ R\ \} \qquad \mu \vdash_r \{\ R \wedge \neg inj_o(b) \wedge inj_r(b)\ \}\ s\ \{\ R\ \} \end{array}}{\mu \vdash \{\ R\ \}\ \texttt{while}\ r\ R\ s\ \{\ R \wedge \neg inj_o(b) \wedge \neg inj_r(b)\ \}}$$

Figure 4-4: Lockstep Control Flow and Structural Rules

4.2 Proof Rules

Figures 4-2, 4-3, and 4-4 provide an abbreviated presentation of the rules of my program logic. I have partitioned the presentation into three parts:

- The *left rules* and *right rules* for primitive statements.
- The left rules for control flow.
- The *lockstep* rules.

4.2.1 Left and Right Rules for Primitive Statements

Figure 4-2 presents the *left rules* and *right* rules of Leto's relational Hoare logic. The *left rules* with the notation $\vdash_l \{ RP \} \; s \; \{ RQ \}$ characterize the behavior of the reliable execution of the statement s. The *right rules* with the notation $\mu \vdash_r \{ RP \} \; s \; \{ RQ \}$ characterize the behavior of the relaxed execution of s under an execution model specification μ.

The denotation of the left rule notation $\vdash_l \{ RP \} \; s \; \{ RQ \}$ is that if $(\varepsilon_1, \varepsilon_2) \vDash RP$, and $\langle s, \varepsilon_1 \rangle \Downarrow \varepsilon_1'$, then $(\varepsilon_1', \varepsilon_2) \vDash RQ$. Namely, given a proof in the left rules, for a pair of environments satisfying the precondition of the proof, then if a reliable execution of s terminates, then the resulting environment pair satisfies the proof's postcondition.

The denotation of the right rule notation $\mu \vdash_r \{ RP \} \; s \; \{ RQ \}$ is similar to that of the left rule notation: if $(\varepsilon_1, \varepsilon_2) \vDash RP$, $\langle s, \varepsilon_2 \rangle \Downarrow_\mu \varepsilon_2'$, then $(\varepsilon_1, \varepsilon_2') \vDash RQ$. Namely, given a proof in the right rules, for a pair of environments satisfying the proof's precondition, then if execution of s under the execution model specification μ terminates, then the resulting environment pair satisfies the proof's postcondition.

Register Assignment. The register assignment statement $r = n$ in the lowered language is reliable in both the reliable and relaxed executions. In the reliable execution, the rule [ASSIGN-L] captures the semantics of the assignment statement via the standard backward characterization of assignment as seen in standard Hoare logic [19]. The major distinction between a standard presentation and the presentation here is that the substitution replaces the injected form of the register r in the postcondition of the statement. The expression

$inj_o(r)$ denotes the value of r in the reliable version of the program. For the relaxed execution, the rule [ASSIGN-R] captures the semantics by substituting for $inj_r(r)$, which denotes the value of r in the relaxed execution.

Arithmetic Operation. The rules [BINOP-L] and [BINOP-R] give the semantics of binary arithmetic operations on registers: $r = r_1 \oplus r_2$. For the reliable execution,[BINOP-L] relies on the backwards characterization of assignment as seen in [ASSIGN-L] to substitute the value r in the reliable execution of the program with the value of the arithmetic operation $inj_o(r_1 \oplus r_2)$.

For the relaxed execution, [BINOP-R], augments the traditional backwards characterization to include the potentially unreliable execution of the binary operation. The rule incorporates the unreliable execution into the semantics by substituting for the destination register a new fresh register r'. The rule then constrains the value of r' such that, conditional on the current state of the execution model, the register satisfies the postconditions that may result from each enabled version of the operation.

Read. The rules [READ-L] and [READ-R] give the semantics of reads from memory. The left rule [READ-L] mimics the behavior of [READ-L] with the primary differing being that it substitutes the value of a register, r, for a local variable x. The right rule, [READ-R], on the other hand more closely resembles [BINOP-R] in that it models the potentially unreliable execution of the read from memory.

Write. The rules [WRITE-L] and [WRITE-R] give the semantics of writes to memory. These rules are analogous to [READ-L] and [READ-R], except modified in their exact implementation to captures writes to memory.

Assert. The rules [ASSERT-L] and [ASSERT-R] give the semantics of assertion statements. There is a major distinction between the role of assertion statements between the reliable and relaxed execution of the program. Namely, while the logic requires the condition of an assert statement is verified in the relaxed execution, the condition of an assert statement in the reliable execution does not need to be verified; it is instead assumed. The major design

point is that Leto enables a developer to use a variety of means (e.g., testing, verification, or code review) to validate an assertion in the original program and transfer that reasoning to the verification process for the relaxed execution. To achieve this design, the left rule for assertions assumes the validity of the assertion whereas the right rule asserts.

Assume. The rules [ASSUME-L] and [ASSUME-R] give the semantics of assume statements. The primary distinction for `assume` statements is that while assume statements have their standard semantics in the reliable execution of the program (no proof obligation is required), `assume` statements do in fact require a proof obligation in the relaxed semantics. The semantics of an `assume` statement in the relaxed semantics is therefore the same as that of an `assert` statement. The rationale behind this design is that as part of the verification of the relaxed execution I must verify that faults do not interfere with the reasoning behind an assumption.

Relational Assert The rules [REALATIONAL-ASSERT-L] and [RELATIONAL-ASSERT-R] give the semantics of relational assertion statements. These statements are similar to my normal assert statements, but they take relational predicates as arguments rather than standard predicates. The logic requires that the condition of a relational assertion is verified in relaxed execution, but in reliable execution the logic treats these statements as no-ops.

4.2.2 Left and Right Rules for Control Flow

Figure 4-3 presents the left and right rules for control flow statements. With the exception of [WHILE-L], the rules adhere to the standard formalization as seen in traditional Hoare logic. The only distinction between these rules and their standard implementation is that they operate over relational predicates.

The rule [WHILE-L] assumes that $RP \wedge \neg b$ holds after the while loop and does not place any constraints on the environment prior to the loop. In other words, I do not verify the invariants on loops that run under left semantics.

4.2.3 Lockstep Rules for Control Flow

Figure 4-4 presents the lockstep rules for control flow statements. The lockstep rules together constitute the main top-level judgment of the logic: $\mu \vdash \{\ RP\ \}\ s\ \{\ RQ\ \}$. The denotation of the judgment is that if $(\varepsilon_1, \varepsilon_2) \models RP$, $\langle s, \varepsilon_1 \rangle \Downarrow \varepsilon_1'$, and $\langle s, \varepsilon_2 \rangle \Downarrow_\mu \varepsilon_2'$, then $(\varepsilon_1', \varepsilon_2') \models RQ$. I label this judgment the lockstep judgment because it reasons about relations between the two semantics as they proceed in lockstep, a single statement at a time.

Stage. The rule [STAGE] gives a semantics to a pair of statements s_1 and s_2 for which the goal is to characterize the behavior when the reliable execution executes s_1 and the relaxed execution executes s_2. The specific composition I have chosen for this rule is to apply the left rules for s_1 before applying the right rules to s_2. Namely,the rule first applies the left rule for s_1, yielding a new predicate RR, before then applying the right rule for s_2 to RR. The rule [SPLIT] provides a rationale for this specific composition.

Split. The rule [SPLIT] gives a semantics to individual statements in the lockstep semantics. The rule relies on the [STAGE] rule to apply the left rules for the statement before applying the right rules. This design forces a specific composition of the rules in order to achieve more tractable verification. For example, for a statement `assert` r, this rule will first apply the left rule for assertions, which can be used to derive $r\texttt{<o>} = \mathit{true}$. Note that this derivation occurs by assumption as the logic assumes the validity of assertions in the reliable execution. Next, the rule requires the proof establish that $r\texttt{<r>} = \mathit{true}$. If, for example, the predicate $r\texttt{<o>} = r\texttt{<r>}$ is in the context, then this proof obligation is easily established.

Sequential Composition. The rule [SEQ] gives the standard semantics as found in the standard Hoare Logic for sequential composition with the distinction that it operates on relational predicates.

Weakening. The rule [WEAK] gives the standard semantics for weakening as found in the standard Hoare logic with the distinction that it operates on relational predicates.

If. The rule [IF] gives the semantics of `if` statements. The rule considers four cases:

- The register r evaluates to *true* in both the reliable and relaxed execution.
- r evaluates to *true* in the reliable semantics but *false* in the relaxed semantics.
- r evaluates to *false* in the reliable semantics but *true* in the relaxed semantics.
- r evaluates to *false* in both the reliable and relaxed semantics.

For the first and fourth cases, execution proceeds in lockstep fashion and therefore the rule recurses, adding the determined validity of r in each of the semantics to the precondition for each recursion. For these cases, the rule mimics that of a standard Hoare Logic.

For the second and third cases, the logic must capture the fact that the two executions have diverged. The logic captures this by leveraging the staging rule to apply the left rules for the branch on which the reliable execution has taken before considering the branch on which the relaxed version has taken. Again, this forces a specific methodology for reasoning about the programs in that the logic extracts the full availability of assertions that may exist on the branch that the reliable execution takes before proceeding with the relaxed execution.

While. The rule [WHILE] gives the semantics of `while` statements. The rule is similar in design to the rule for `if` statements in that it must also consider cases in which the control flow of the two executions diverge. The rule first considers the case where the two executions proceed in lockstep by both executing an iteration of the loop. The next two cases leverage the left and right rules to consider the cases when:

- The relaxed execution halts, but the reliable execution executes an additional iteration.
- The reliable execution halts, but the relaxed execution executes an additional iteration, respectively.

4.3 Properties

Leto's program logic ensures two basic properties of Leto programs: *preservation* and *progress*.

Lemma 4.1 (Left Preservation).
If $\vdash_l \{ RP \} \; s \; \{ RQ \}$ *and* $(\varepsilon_1, \varepsilon_2) \models RP$ *and* $\langle s, \varepsilon_1 \rangle \Downarrow \varepsilon_1'$*, then* $(\varepsilon_1', \varepsilon_2) \models RQ$.

Leto's left preservation property states that given a proof in the left rules, for a pair of environments satisfying the precondition of the proof, then if a reliable execution of s terminates, then the resulting environment pair satisfies the proof's postcondition.

Proof. By induction on the lemma statement:

- ASSIGN-L. Stepping $\langle s, \varepsilon_1 \rangle$ produces an environment ε_1' which differs from ε_1 only in that r is mapped to n in σ. As $\varepsilon_1 \models RQ[n/inj_o(r)]$, ε_1' trivially satisfies RQ. Additionally, as the precondition contains no substitutions on $inj_r(r)$, ε_2 trivially satisfies RQ.

- BINOP-L. Similar to ASSIGN-L.

- READ-L. Similar to ASSIGN-L.

- WRITE-L. Similar to ASSIGN-L.

- ASSERT-L. Stepping $\langle s, \varepsilon_1 \rangle$ by definition adds r to ε_1 to form ε_1'. Therefore, $(\varepsilon_1', \varepsilon_2) \models inj_o(r)$.

- ASSUME-L. Similar to ASSERT-L.

- RELATIONAL-ASSERT-L. Relational assertions in left mode do nothing.

- SEQ-L. I start with inversion on $\vdash_l \{ RP \} \ s \ \{ RQ \}$, which yields $\vdash_l \{ RP \} \ s_1 \ \{ RR \}$ and $\vdash_l \{ RR \} \ s_2 \ \{ RQ \}$. Applying the induction hypothesis to $\vdash_l \{ RP \} \ s_1 \ \{ RR \}$ yields $(\varepsilon_1'', \varepsilon_2) \models RR$. Applying the induction hypothesis to $\vdash_l \{ RR \} \ s_2 \ \{ RQ \}$ yields $(\varepsilon_1', \varepsilon_2) \models RQ$.

- WHILE-L. Stepping $\langle s, \varepsilon_1 \rangle$ by definition adds RP to ε_1 to form ε_1'. Therefore, $(\varepsilon_1', \varepsilon_2) \models RP$.

- IF-L. First, I perform inversion on s. Then, I destruct b. In the case where b is *true*, I apply the induction hypothesis to $\vdash_l \{ RP \land b \} \ s \ \{ RQ \}$. In the case where b is *false*, I apply the induction hypothesis to $\vdash_l \{ RP \land \neg b \} \ s \ \{ RQ \}$.

□

Lemma 4.2 (Right Preservation).
If $\mu \vdash_r \{ RP \}\ s\ \{ RQ \}$ *and* $(\varepsilon_1, \varepsilon_2) \models RP$ *and* $\langle s, \varepsilon_2 \rangle \Downarrow_\mu \varepsilon_2'$, *then* $(\varepsilon_1, \varepsilon_2') \models RQ$.

Leto's right preservation property states that given a proof in the right rules, for a pair of environments satisfying the precondition of the proof, then if execution of s under the execution model specification μ terminates, then the resulting environment pair satisfies the proof's postcondition.

Proof. By induction on the lemma statement:

- ASSIGN-R. Stepping $\langle s, \varepsilon_2 \rangle$ produces an environment ε_2' which differs from ε_2 only in that r is mapped to n in σ. As $\varepsilon_1 \models RQ[n/inj_o(r)]$, ε_2' trivially satisfies RQ. Additionally, as the precondition contains no substitutions on $inj_r(r)$, ε_1 trivially satisfies RQ.

- BINOP-R. Stepping $\langle s, \varepsilon_2 \rangle$ produces an environment ε_2' which differs from ε_2 only in that r is mapped to r' in σ. The restrictions placed on r' in RQ' codify the operator substitution routine from F-BINOP in the operational semantics. Therefore, the runtime always sets r in a way such that the resulting environment satisfies RQ. Additionally, as the precondition contains no substitutions on $inj_r(r)$, ε_1 trivially satisfies RQ.

- READ-R. Similar to BINOP-R.

- WRITE-R. Similar to BINOP-R.

- ASSERT-R. Stepping $\langle s, \varepsilon_2 \rangle$ does not modify ε_2. Therefore, $\varepsilon_2 = \varepsilon_2'$. Since the precondition and postcondition are identical, $(\varepsilon_1, \varepsilon_2') \models r$.

- ASSUME-R. Similar to ASSERT-R.

- RELATIONAL-ASSERT-R. Similar to ASSERT-R.

- SEQ-R. I start with inversion on $\vdash_r \{ RP \}\ s\ \{ RQ \}$, which yields $\vdash_r \{ RP \}\ s_1\ \{ RR \}$ and $\vdash_r \{ RR \}\ s_2\ \{ RQ \}$. Applying the induction hypothesis to $\vdash_r \{ RP \}\ s_1\ \{ RR \}$ yields $(\varepsilon_1, \varepsilon_2'') \models RR$. Applying the induction hypothesis to $\vdash_r \{ RR \}\ s_2\ \{ RQ \}$ yields $(\varepsilon_1, \varepsilon_2') \models RQ$.

- WHILE-R. First, I perform inversion on s. Then, I destruct b. In the case where b is *true*, I apply the induction hypothesis to $\vdash_r \{ RP \wedge b \} s \{ RP \}$. This proves that RP holds after each loop iteration. Upon exiting the loop, $\neg b$ trivially holds. In the case where b is *false*, the loop does not run and therefore $(\varepsilon_1, \varepsilon_2')$ trivially satisfies $RP \wedge \neg b$.

- IF-R. First, I perform inversion on s. Then, I destruct b. In the case where b is *true*, I apply the induction hypothesis to $\vdash_r \{ RP \wedge b \} s \{ RQ \}$. In the case where b is *false*, I apply the induction hypothesis to $\vdash_r \{ RP \wedge \neg b \} s \{ RQ \}$.

□

Theorem 4.1 (Preservation).
If $\mu \vdash \{ RP \} s \{ RQ \}$ *and* $(\varepsilon_1, \varepsilon_2) \models RP$ *and* $\langle s, \varepsilon_1 \rangle \Downarrow \varepsilon_1'$ *and* $\langle s, \varepsilon_2 \rangle \Downarrow_\mu \varepsilon_2'$*, then* $(\varepsilon_1', \varepsilon_2') \models RQ$

Leto's preservation property states that given a proof in the program logic of a program s, for all pairs of environments $(\varepsilon_1, \varepsilon_2)$ that satisfy the proof's precondition, if the executions of s under both the reliable semantics and the relaxed semantics terminate in a pair of environments $(\varepsilon_1', \varepsilon_2')$, then this pair of environments satisfies the proof's postcondition. Note that this states the partial correctness of the logic and does not establish termination (and therefore total correctness).

Proof. By induction on the theorem statement:

- STAGE. I first perform inversion on s. Then, I apply Lemma 4.1 to $\vdash_l \{ RP \} s_1 \{ RR \}$ and Lemma 4.2 to $\vdash_r \{ RR \} s_2 \{ RQ \}$.

- SPLIT. I first perform inversion on s, followed by inversion on $\mu \vdash \{ RP \} s \sim s \{ RQ \}$. Then, I apply Lemma 4.1 to $\vdash_l \{ RP \} s \{ RR \}$ and Lemma 4.2 to $\vdash_r \{ RR \} s \{ RQ \}$.

- SEQ. I start with inversion on $\mu \vdash \{ P \} s \{ Q \}$, which yields $\mu \vdash \{ P \} s_1 \{ R \}$ and $\mu \vdash \{ R \} s_2 \{ Q \}$. Applying the induction hypothesis to $\mu \vdash \{ P \} s_1 \{ R \}$ yields $(\varepsilon_1'', \varepsilon_2'') \models R$. Applying the induction hypothesis to $\mu \vdash \{ R \} s_2 \{ Q \}$ yields $(\varepsilon_1', \varepsilon_2') \models Q$.

- WEAK. I start with inversion on $\mu \vdash \{ RP \} s \{ RQ \}$. Applying the induction hypothesis to $\mu \vdash \{ RP' \} s \{ RQ' \}$ yields $(\varepsilon_1', \varepsilon_2') \models RQ'$. Since $RQ' \models RQ$, $(\varepsilon_1', \varepsilon_2') \vdash RQ$.

- IF. I begin with inversion on s. Then, I destruct b. In the case where $inj_o(b) \wedge inj_r(b)$, I apply the induction hypothesis to $\mu \vdash \{ P \wedge inj_o(b) \wedge inj_r(b) \} s_1 \{ Q \}$. In the case where $\neg inj_o(b) \wedge inj_r(b)$, I apply the induction hypothesis to

$$\mu \vdash \{ P \wedge \neg inj_o(b) \wedge inj_r(b) \} s_2 \sim s_1 \{ Q \}.$$

In the case where $inj_o(b) \wedge \neg inj_r(b)$, I apply the induction hypothesis to

$$\mu \vdash \{ P \wedge inj_o(b) \wedge \neg inj_r(b) \} s_1 \sim s_2 \{ Q \}.$$

In the case where $\neg inj_o(b) \wedge \neg inj_r(b)$, I apply the induction hypothesis to

$$\mu \vdash \{ P \wedge \neg inj_o(b) \wedge \neg inj_r(b) \} s_2 \{ Q \}.$$

- WHILE. First, I perform inversion on s. Then, I destruct b. In the case where $inj_o(b) \wedge inj_r(b)$, I apply the induction hypothesis to $\mu \vdash \{ R \wedge inj_o(b) \wedge inj_r(b) \} s \{ R \}$. This proves that R holds after each loop iteration. Upon exiting the loop, $\neg inj_o(b) \wedge \neg inj_r(b)$ trivially holds. The cases where $\neg inj_o(b) \wedge inj_r(b)$ and $inj_o(b) \wedge \neg inj_r(b)$ are similar to the previous case. In the case where $\neg inj_o(b) \wedge \neg inj_r(b)$, the loop does not run in either execution and therefore $(\varepsilon_1', \varepsilon_2')$ trivially satisfies the postcondition.

□

Lemma 4.3 (Right Progress).

If $\mu \vdash_r \{ RP \} s \{ RQ \}$ *and* $(\varepsilon_1, \varepsilon_2) \models RP$ *and* $\langle s, \varepsilon_2 \rangle \Downarrow_\mu \varepsilon_2'$, *then* $\neg failed(\varepsilon_2')$ *where* $failed(\langle \mathtt{fail}, \varepsilon \rangle) = true$

Leto's right progress property states that given a proof in the right rules of a program s, for all pairs of environments $(\varepsilon_1, \varepsilon_2)$ that satisfy the proof's precondition, then if the relaxed execution of s under μ terminates, then it does not terminate in an error. The right progress property establishes that a Leto's program of right rules satisfies all of its `assert` and `assume` statements.

Proof. By induction on the theorem statement:

- ASSIGN-R. Assignment cannot fail.
- BINOP-R. Binary operations cannot fail.
- READ-R. Reads cannot fail.
- WRITE-R. Writes cannot fail.
- ASSERT-R. Stepping $\langle s, \varepsilon_2 \rangle$ produces the environment ε_2' where $\varepsilon_2 = \varepsilon_2'$. Since $\neg failed(\varepsilon_2)$, $\neg failed(\varepsilon_2')$ trivially holds.
- ASSUME-R. Similar to ASSERT-R.
- RELATIONAL-ASSERT-R. Similar to ASSERT-R.
- SEQ-R. I start with inversion on $\vdash_r \{ RP \}\ s\ \{ RQ \}$, which yields $\vdash_r \{ RP \}\ s_1\ \{ RR \}$ and $\vdash_r \{ RR \}\ s_2\ \{ RQ \}$. Applying Lemma 4.2 to $\vdash_r \{ RP \}\ s_1\ \{ RR \}$ yields $(\varepsilon_1, \varepsilon_2'') \vDash RR$. Applying the induction hypothesis to $\vdash_r \{ RP \}\ s_1\ \{ RR \}$ provides $\neg failed(\varepsilon_2'')$. Applying the induction hypothesis to $\vdash_r \{ RR \}\ s_2\ \{ RQ \}$ yields $\neg failed(\varepsilon_2')$.
- WHILE-R. First, I perform inversion on s. Then, I destruct b. In the case where b is *true*, I apply the induction hypothesis to $\vdash_r \{ RP \wedge b \}\ s\ \{ RP \}$.

  In the case where b is *false*, the loop does not run and therefore $(\varepsilon_2 = \varepsilon_2')$ trivially satisfies $\neg failed(\varepsilon_2')$.

  Lastly, since $(\varepsilon_1, \varepsilon_2) \vDash RP$, the invariant check before the loop cannot fail.
- IF-R. First, I perform inversion on s. Then, I destruct b. In the case where b is *true*, I apply the induction hypothesis to $\vdash_r \{ RP \wedge b \}\ s\ \{ RQ \}$. In the case where b is *false*, I apply the induction hypothesis to $\vdash_r \{ RP \wedge \neg b \}\ s\ \{ RQ \}$.

□

Theorem 4.2 (Progress).

If $\mu \vdash \{ RP \}\ s\ \{ RQ \}$ *and* $(\varepsilon_1, \varepsilon_2) \models RP$ *and* $\langle s, \varepsilon_1 \rangle \Downarrow \varepsilon_1'$ *and* $\langle s, \varepsilon_2 \rangle \Downarrow_\mu \varepsilon_2'$, *then* $\neg failed(\varepsilon_2')$ *where* $failed(\langle \boldsymbol{fail}, \varepsilon \rangle) = true$

Leto's progress property states that given a proof in the program logic of a program s, for all pairs of environments $(\varepsilon_1, \varepsilon_2)$ that satisfy the proof's precondition, if the reliable execution of s terminates successfully, then if the relaxed execution of s under μ terminates, then it does not terminate in an error. The progress property establishes that a Leto's program satisfies all of its `assert` and `assume` statements – provided that all reliable executions of the program also satisfy the program's `assert` and `assume` statements.

Proof. By induction on the theorem statement:

- STAGE. I first perform inversion on s. Then, I apply Lemma 4.1 to $\vdash_l \{\ RP\ \}\ s_1\ \{\ RR\ \}$ and Lemma 4.3 to $\vdash_r \{\ RR\ \}\ s_2\ \{\ RQ\ \}$.

- SPLIT. I first perform inversion on s, followed by inversion on $\mu \vdash \{\ RP\ \}\ s \sim s\ \{\ RQ\ \}$. Then, I apply Lemma 4.1 to $\vdash_l \{\ RP\ \}\ s_1\ \{\ RR\ \}$. Finally, I apply Lemma 4.3 to $\vdash_r \{\ RR\ \}\ s_2\ \{\ RQ\ \}$.

- SEQ. I start with inversion on $\mu \vdash \{\ P\ \}\ s\ \{\ Q\ \}$, which yields $\mu \vdash \{\ P\ \}\ s_1\ \{\ R\ \}$ and $\mu \vdash \{\ R\ \}\ s_2\ \{\ Q\ \}$. Applying the induction hypothesis to $\mu \vdash \{\ P\ \}\ s_1\ \{\ R\ \}$ yields $\neg \mathit{failed}(\varepsilon_2'')$. Applying the induction hypothesis to $\mu \vdash \{\ R\ \}\ s_2\ \{\ Q\ \}$ yields $\neg \mathit{failed}(\varepsilon_2')$.

- WEAK. I start with inversion on $\mu \vdash \{\ RP\ \}\ s\ \{\ RQ\ \}$. Applying the induction hypothesis to $\mu \vdash \{\ RP'\ \}\ s\ \{\ RQ'\ \}$ yields $\neg \mathit{failed}(\varepsilon_2'')$. Since $RP \models RP'$ and $RQ' \models RQ$, $\neg \mathit{failed}(\varepsilon_2'') \rightarrow \neg \mathit{failed}(\varepsilon_2')$.

- IF. I begin with inversion on s. Then, I destruct b. In the case where $\mathit{inj}_o(b) \wedge \mathit{inj}_r(b)$, I apply the induction hypothesis to $\mu \vdash \{\ P \wedge \mathit{inj}_o(b) \wedge \mathit{inj}_r(b)\ \}\ s_1\ \{\ Q\ \}$. In the case where $\neg \mathit{inj}_o(b) \wedge \mathit{inj}_r(b)$, I apply the induction hypothesis to

  $$\mu \vdash \{\ P \wedge \neg \mathit{inj}_o(b) \wedge \mathit{inj}_r(b)\ \}\ s_2 \sim s_1\ \{\ Q\ \}.$$

  In the case where $\mathit{inj}_o(b) \wedge \neg \mathit{inj}_r(b)$, I apply the induction hypothesis to

  $$\mu \vdash \{\ P \wedge \mathit{inj}_o(b) \wedge \neg \mathit{inj}_r(b)\ \}\ s_1 \sim s_2\ \{\ Q\ \}.$$

In the case where $\neg inj_o(b) \wedge \neg inj_r(b)$, I apply the induction hypothesis to

$$\mu \vdash \{ \ P \wedge \neg inj_o(b) \wedge \neg inj_r(b) \ \} \ s_2 \ \{ \ Q \ \}.$$

- WHILE. First, I perform inversion on s. Then, I destruct b. In the case where $inj_o(b) \wedge inj_r(b)$, I apply the induction hypothesis to $\mu \vdash \{ \ R \wedge inj_o(b) \wedge inj_r(b) \ \} \ s \ \{ \ R \ \}$.

  In the case where $inj_o(b) \wedge \neg inj_r(b)$, stepping $\vdash_l \{ \ R \wedge inj_o(b) \wedge \neg inj_r(b) \ \} \ s \ \{ \ R \ \}$ yeilds ε_2' where $\varepsilon_2' = \varepsilon_2$. Since $\neg failed(\varepsilon_2)$, $\neg failed(\varepsilon_2')$ holds as well.

  In the $\neg inj_o(b) \wedge inj_r(b)$ case, I apply Lemma 4.3 to $\mu \vdash_r \{ \ R \wedge \neg inj_o(b) \wedge inj_r(b) \ \} \ s \ \{ \ R \ \}$.

  In the case where $\neg inj_o(b) \wedge \neg inj_r(b)$, the loop doesn't run and therefore $\varepsilon_2 = \varepsilon_2'$, so $\neg failed(\varepsilon_2')$ trivially holds.

  Lastly, since $(\varepsilon_1, \varepsilon_2) \vDash R$, the invariant check before the loop cannot fail.

$\square$

Chapter 5

Verification Algorithm

Figures 5-1, 5-2, and 5-3 present the core of Leto's verification algorithm. The algorithm performs forward symbolic execution to discharge verification conditions generated by `assert`, `assert_t`, `invariant`, and `invariant_r` statements in the program. The algorithm directly implements the Hoare-style relational program logic [12] I presented in Chapter 4.

However, unlike the program logic, I present this algorithm over expressions rather than registers. To translate from the expression representation to the register representation, use the following algorithm:

1. Enumerate all expressions $e_1, \ldots, e_n$.
2. Immediately before the evaluation of e_i, construct a register r_i and store the result of evaluating e_i in r_i.
3. Replace e_i in the program with r_i.

To translate from the register representation to the expression representation, use the following algorithm:

1. Enumerate all registers $r_1, \ldots, r_n$.
2. For each register r_i, generate a variable v_i in the reliable memory region.
3. For each register r_i, replace r_i with v_i.

Preliminaries. I denote Leto's verification algorithm by the function Ψ, which takes as input a statement s, a list of logical predicates, σ, a model specification m, and a verification mode c. The statement s is the statement to be verified, σ is the symbolic context under which the verification algorithm is invoked, m computes the symbolic representation for an operation in the execution model.

The control value $c \in C = \{lock, left, right\}$ determines whether or not the algorithm is performing verification in *lockstep* mode, *left mode*, or *right mode*, respectively. When performing verification in lockstep, the algorithm models the original and relaxed execution and executes each one an instruction at a time. In this mode, the algorithm is able to demonstrate an easy correspondence between the two executions that therefore enables the algorithm to, for example, transfer assumed properties of the original execution over to verify the relaxed execution. For both left mode and right mode, the algorithm assumes the two executions have diverged and, therefore, that there is no simple correspondence between the two executions. In left mode, the algorithm symbolically evaluates the original execution of the program, ignoring the verification conditions required for the relaxed execution. In right mode, the algorithm symbolically evalutes the relaxed execution of the program and checks the verification conditions that are required of the relaxed execution.

The function $\text{vexp}(e, m, c)$ maps a standard unary expression e to a list of constraints that represent the resulting value in either the original or relaxed execution. For example, vexp maps a variable reference x to either the variable reference x<o> or x<r> if c equals `left` or `right`, respectively. The function returns a list of constraints because the expression may reference an operation suffixed with a period, denoting that the operation has a custom semantics. The list of constraints characterizes the non-deterministic choice of the operation's implementation. The function uses the model specification m to compute the symbolic representation for these operations.

The function $\text{vbexp}(b, m, c)$ maps a standard unary boolean expression b to a list of constraints. Its operation is similar to that of vexp.

Assignment. For an assignment statement x=e, the algorithm maps the e to an appropriate relational expression for both the original and relaxed execution by creating the constraint

1: **function** $\Psi(s, \sigma, m, c)$
2: **match** s **do**
3: $x = e$:
4: $\sigma_o \leftarrow (x\text{<o>} = \text{vexp}(e, m, \mathit{left}))$
5: $\sigma_r \leftarrow (x\text{<r>} = \text{vexp}(e, m, \mathit{right}))$
6: **return** $\sigma :: \text{join}(c, \sigma_o, \sigma_r)$
7: `assert` b_v:
8: `assume` b_v:
9: $\sigma_o \leftarrow \text{vbexp}(b_v, m, \mathit{left})$
10: $\sigma_r \leftarrow \text{vbexp}(b_v, m, \mathit{right})$
11: **if** $(c == \mathit{lock})$ **then**
12: $\text{Verify}(\sigma :: \sigma_o, \sigma_r)$
13: **else if** $(c == \mathit{right})$ **then**
14: $\text{Verify}(\sigma, \sigma_r)$
15: **end if**
16: **return** $\sigma :: \text{join}(c, \sigma_o, \sigma_r)$
17: `assert_r` (b_r):
18: $\text{Verify}(\sigma, b_r)$
19: **return** $\sigma :: b_r$
20: $s1; s2$:
21: **return** $\Psi(s_2, \sigma :: \Psi(s_1, \sigma, m, c), m, c)$
22: **end match**
23: **end function**

Figure 5-1: Verification Algorithm (sans Control Flow)

that x in the original (relaxed) execution has the value e. The algorithm then uses the join$(\cdot)$ function to return a result. The join$(\cdot)$ function joins two constraints into a list depending on the value of c. If $c = \mathit{lock}$ – denoting the the algorithm is modelling the lockstep execution of both the original and relaxed executions – then the *join* includes both constraints. If $c = \mathit{left}$ or $c = \mathit{right}$ – denoting that the algorithm is modeling the original or relaxed execution, respectively – then *join* includes only the first or second constraint, respectively.

Assume and Assert. The algorithm verifies both `assert` and `assume` using the same logical approach. The algorithm first generates the verification conditions for both the original and relaxed executions, namely that the statement's boolean expression b_v is true (σ_o and σ_r, respectively). The algorithm next considers two cases. In lockstep mode, the algorithm verifies that the current context σ extended with the *assumption* that the assertion or assumption is true in the original execution implies that the verification condition holds. The function Verify(σ_1, σ_2) verifies that σ_1 implies σ_2 (Leto specifically uses an SMT solver to do so) and halts the execution of the algorithm if the implication does not hold or the solver is unable to demonstrate that it holds. In right mode, the the algorithm directly verifies that the current context implies the verification condition. The insight is that unlike in lockstep mode, the algorithm must verify the relaxed execution independently of the original execution and, therefore, the algorithm cannot leverage the assumption that the assertion or assumption is valid in the original execution. In the last step, the algorithm returns the join of the two verification conditions.

```
1: function Ψ(if (b) {s1} else {s2}, σ, m, c) =
2:     σo ← vbexp(b, m, left),  σr ← vbexp(b, m, right)
3:     if (c == lock) then
4:         σ1 ← Ψ(s1, σ :: σo :: σr, lock)
5:         σ2 ← Ψ(s2, σ :: ¬σo :: ¬σr, lock)
6:
7:         σ3 ← Ψ(s1, σ :: σo :: ¬σr, left)
8:         σ4 ← Ψ(s2, σ :: σ3 :: σo :: ¬σr, right)
9:
10:        σ5 ← Ψ(s2, σ :: ¬σo :: σr, left)
11:        σ6 ← Ψ(s1, σ :: σ5 :: ¬σo :: σr, right)
12:
13:        return (σ1 :: σ2 :: σ4 :: σ6)
14:    else if (c == left) then
15:        return Ψ(s1, σ :: σo, left)::Ψ(s2, σ :: ¬σo, left)
16:    else if (c == right) then
17:        return Ψ(s1, σ :: σr, right)::Ψ(s2, σ :: ¬σr, right)
18:    end if
19: end function
```

Figure 5-2: If statement verification algorithm

Relational Assert. The algorithm verifies relational assertions under the current context. If verification fails, then the verification procedure halts. If verification succeeds, then the algorithm appends the assertion to the context and returns the result.

If. Figure 5-2 presents the algorithm's implementation for `if` statement verification. The algorithm has a different implementation for each of the verification modes:

- **Lockstep**. In lockstep mode, the algorithm verifies and generates a symbolic representation for four different scenarios:
  - The original execution and relaxed execution both take the *true* branch of the statement, represented by σ_1.
  - The original execution and relaxed execution both take the *false* branch of the statement, σ_2.
  - The original execution takes the *true* branch and the relaxed execution takes the *false* branch, σ_4.

```
1: function Ψ(while (b) (b_v) (b_r) {s_b}, σ, m, c) =
2:     σ_o ← vbexp(b, m, left),   σ_r ← vbexp(b, m, right)
3:     if (c == lock) then
4:         p_o ← vbexp(b_v, m, left),   p_r ← vbexp(b_v, m, right)
5:         Verify(σ :: p_o, p_r :: b_r)
6:         σ_c ← σ :: p_o :: p_r :: b_r
7:         Verify(Ψ(s_b, σ_c :: σ_o :: σ_r, m, lock), b_r :: p_o :: p_r)
8:         Verify(Ψ(s_b, σ_c :: ¬σ_o :: σ_r, m, right), b_r :: p_r)
9:         return (σ_c :: ¬σ_o :: ¬σ_r)
10:    else if (c == left) then
11:        return σ :: (vbexp(b_v, m, left) :: ¬σ_o)
12:    else if (c == right) then
13:        p ← vbexp(b_v, m, right)
14:        Verify(σ, p :: b_r)
15:        Verify(Ψ(s_b, σ :: p :: b_r :: σ_r, m, right), p :: b_r)
16:        return (σ :: p :: b_r :: ¬σ_r)
17:    end if
18: end function
```

Figure 5-3: While statement verification algorithm

  - The original execution takes the *false* branch and the relaxed execution takes the *true* branch, σ_6.

- **Left**. In left mode, the algorithm need only generate a symbolic representation for the original execution. The algorithm achieves this by conjoining the results of recursive calls to Ψ on s_1 and s_2 given the current context.

- **Right**. In right mode, the algorithm need only generate a symbolic representation and discharge the verification conditions for the relaxed execution. Similar to that of left mode, algorithm achieves this by conjoining the results of recursive calls to Ψ on s_1 and s_2.

While. Figure 5-3 presents the algorithm for verifying `while` loops. The algorithm has a different implementation for each of the verification modes:

- **Lockstep**. In lockstep mode, the algorithm considers two cases:

  - Both the original and relaxed execution take a step forward together in lockstep (Line 7).

  - The original execution has finished executing the loop and the relaxed execution continues its execution of the loop (Line 8).

  The implementation first verifies the loop invariants hold given the context (Line 5), then verifies that after symbolically executing each of the two execution cases that the loop invariants hold afterwards.

- **Left**. In left mode, the algorithm returns the symbolic representation of the original execution of the loop. Because the algorithm assumes that the original program has been verified to be correct, the resulting symbolic representation is simply the loop invariant.

- **Right**. In right mode, the algorithm returns the symbolic representation of the relaxed execution of the loop. Unlike the original execution, this case requires verifying that the loop invariant holds at the beginning of the loop (Line 13) and that it holds after each iteration (Line 15).

5.1 Invariant Inference

Figures 5-4 and 5-5 present Leto's loop invariant inference algorithm. Leto uses Houdini-style loop invariant inference [17] to reduce the annotation burden on the programmer.

Preliminaries I denote a modified version of Leto's verification algorithm by the function Ψ', which is identical to Ψ except that it ignores calls to the Verify function.

The value $r \in R = \{sat, unsat, unknown\}$ represents the response from the SMT solver Leto uses. *sat* indicates that a satisfying assignment exists for all variables in the predicate. *unsat* indicates that no satisfying assignment exists for all variables in the predicate. *unknown* indicates that the SMT solver cannot determine whether a satisfying assignment exists.

The function Check is identical to the Verify function except that:

- Check(σ_1, σ_2) does not halt execution if σ_1 does not imply σ_2.

- Check returns a tuple consisting of:

1: **function** $\textsc{Inf}(\texttt{while}\ (b)\ (b_v)\ (b_r)\ \{s_b\}, \sigma, m, c, b_{pv}, b_{pr}) =$
2: $\quad \sigma_o \leftarrow \text{vbexp}(b, m, \textit{left}),\ \ \sigma_r \leftarrow \text{vbexp}(b, m, \textit{right})$
3: $\quad$ **if** $(c == \textit{lock})$ **then**
4: $\quad\quad p_o \leftarrow \text{vbexp}(b_v, m, \textit{left}),\ \ p_r \leftarrow \text{vbexp}(b_v, m, \textit{right})$
5: $\quad\quad (r_1, b_{fv1}, b_{fr1}) \leftarrow \text{Check}(\sigma :: p_o, p_r :: b_r)$
6: $\quad\quad \sigma_c \leftarrow \sigma :: p_o :: p_r :: b_r :: \text{vbexp}(b_{pv}, m, \textit{left}) :: \text{vbexp}(b_{pr}, m, \textit{right})$
7: $\quad\quad (r_2, b_{fv2}, b_{fr2}) \leftarrow \text{Check}(\Psi'(s_b, \sigma_c :: \sigma_o :: \sigma_r, m, \textit{lock}), b_r :: p_o :: p_r)$
8: $\quad\quad (r_3, b_{fv3}, b_{fr3}) \leftarrow \text{Check}(\Psi'(s_b, \sigma_c :: \neg\sigma_o :: \sigma_r, m, \textit{right}), b_r :: p_r)$
9: $\quad\quad$ **if** $(r_1 == \textit{unknown} \vee r_2 == \textit{unknown} \vee r_3 == \textit{unknown})$ **then**
10: $\quad\quad\quad$ **return** $\textsc{WeakInf}(\texttt{while}\ (b)\ (\textit{true})\ (\textit{true})\ \{s_b\}, \sigma, m, c, p_o :: p_r :: b_r, b_{pv}, b_{pr})$
11: $\quad\quad$ **else if** $(r_1 == \textit{sat} \vee r_2 == \textit{sat} \vee r_3 == \textit{sat})$ **then**
12: $\quad\quad\quad b_{fv} \leftarrow b_{fv1} \cup b_{fv2} \cup b_{fv3}$
13: $\quad\quad\quad b_{fr} \leftarrow b_{fr1} \cup b_{fr2} \cup b_{fr3}$
14: $\quad\quad\quad$ **return** $\textsc{Inf}(\texttt{while}\ (b)\ (b_v \setminus b_{fv})\ (b_r \setminus b_{fr})\ \{s_b\}, \sigma, m, c, b_{pv}, b_{pr})$
15: $\quad\quad$ **else**
16: $\quad\quad\quad$ **return** (b_v, b_r)
17: $\quad\quad$ **end if**
18: $\quad$ **else if** $(c == \textit{right})$ **then**
19: $\quad\quad p \leftarrow \text{vbexp}(b_v, m, \textit{right}),\ \ p_{pv} \leftarrow \text{vbexp}(b_{pv}, m, \textit{right})$
20: $\quad\quad (r_1, b_{fv1}, b_{fr1}) \leftarrow \text{Check}(\sigma, p :: b_r)$
21: $\quad\quad (r_2, b_{fv2}, b_{fr2}) \leftarrow \text{Check}(\Psi'(s_b, \sigma :: p :: b_r :: \sigma_r :: p_{pv}, m, \textit{right}), p :: b_r)$
22: $\quad\quad$ **if** $(r_1 == \textit{unknown} \vee r_2 == \textit{unknown})$ **then**
23: $\quad\quad\quad$ **return** $\textsc{WeakInf}(\texttt{while}\ (b)\ (\textit{true})\ (\textit{true})\ \{s_b\}, \sigma, m, c, p :: b_r, b_{pv}, b_{pr})$
24: $\quad\quad$ **else if** $(r_1 == \textit{sat} \vee r_2 == \textit{sat})$ **then**
25: $\quad\quad\quad b_{fv} \leftarrow b_{fv1} \cup b_{fv2} \cup$
26: $\quad\quad\quad b_{fr} \leftarrow b_{fr1} \cup b_{fr2} \cup$
27: $\quad\quad\quad$ **return** $\textsc{Inf}(\texttt{while}\ (b)\ (b_v \setminus b_{fv})\ (b_r \setminus b_{fr})\ \{s_b\}, \sigma, m, c, b_{pv}, b_{pr})$
28: $\quad\quad$ **else**
29: $\quad\quad\quad$ **return** (b_v, b_r)
30: $\quad\quad$ **end if**
31: $\quad$ **end if**
32: **end function**

Figure 5-4: Strong Inference Algorithm

```
1: function WeakInf(while (b) (b_v) (b_r) {s_b}, σ, m, c, b_cv, b_cr, b_pv, b_pr) =
2:     match b_cv do
3:         b_h :: b'_cv:
4:             if (c == lock) then
5:                 p_o ← vbexp(b_v, m, left) :: vbexp(b_h, m, left)
6:                 p_r ← vbexp(b_v, m, right) :: vbexp(b_h, m, right)
7:                 (r_1, b_fv1, b_fr1) ← Check(σ :: p_o, p_r)
8:                 σ_c ← σ :: p_o :: p_r :: vbexp(b_pv, m, left) :: vbexp(b_pr, m, right)
9:                 (r_2, b_fv2, b_fr2) ← Check(Ψ'(s_b, σ_c :: σ_o :: σ_r, m, lock), p_o :: p_r)
10:                (r_3, b_fv3, b_fr3) ← Check(Ψ'(s_b, σ_c :: ¬σ_o :: σ_r, m, right), p_r)
11:                if (r_1 == unsat ∧ r_2 == unsat ∧ r_3 == unsat) then
12:                    return WeakInf(while (b) (b_h :: b_v) (b_r) {s_b}, σ, m, c, b'_cv, b_cr, b_pv, b_pr)
13:                else return WeakInf(while (b) (b_v) (b_r) {s_b}, σ, m, c, b'_cv, b_cr, b_pv, b_pr)
14:                end if
15:            else if (c == right) then
16:                p ← vbexp(b_v, m, right) :: vbexp(b_h, m, right),  p_pv ← vbexp(b_pv, m, right)
17:                (r_1, b_fv1, b_fr1) ← Check(σ, p)
18:                (r_2, b_fv2, b_fr2) ← Check(Ψ'(s_b, σ :: p :: σ_r :: p_pv, m, right), p)
19:                if (r_1 == unsat ∧ r_2 == unsat) then
20:                    return WeakInf(while (b) (b_h :: b_v) (b_r) {s_b}, σ, m, c, b'_cv, b_cr, b_pv, b_pr)
21:                else return WeakInf(while (b) (b_v) (b_r) {s_b}, σ, m, c, b'_cv, b_cr, b_pv, b_pr)
22:                end if
23:            end if
24:        []:
25:            match b_cr do
26:                []: return (b_v, b_r)
27:                b_h :: b'_cr:
28:                    if (c == lock) then
29:                        p_o ← vbexp(b_v, m, left),  p_r ← vbexp(b_v, m, right)
30:                        (r_1, b_fv1, b_fr1) ← Check(σ :: p_o, b_r :: b_h :: p_r)
31:                        σ_c ← σ :: p_o :: p_r :: b_r :: b_h :: vbexp(b_pv, m, left) :: vbexp(b_pr, m, right)
32:                        (r_2, b_fv2, b_fr2) ← Check(Ψ'(s_b, σ_c :: σ_o :: σ_r, m, lock), p_o :: p_r :: b_r :: b_h)
33:                        (r_3, b_fv3, b_fr3) ← Check(Ψ'(s_b, σ_c :: ¬σ_o :: σ_r, m, right), p_r :: b_r :: b_h)
34:                        if (r_1 == unsat ∧ r_2 == unsat ∧ r_3 == unsat) then
35:                            return WeakInf(while (b) (b_v) (b_r :: b_h) {s_b}, σ, m, c, b_cv, b'_cr, b_pv, b_pr)
36:                        else return WeakInf(while (b) (b_v) (b_r) {s_b}, σ, m, c, b_cv, b'_cr, b_pv, b_pr)
37:                        end if
38:                    else if (c == right) then
39:                        p ← vbexp(b_v, m, right),  p_pv ← vbexp(b_pv, m, right)
40:                        (r_1, b_fv1, b_fr1) ← Check(σ, p :: b_r :: b_h)
41:                        (r_2, b_fv2, b_fr2) ← Check(Ψ'(s_b, σ :: p :: b_r :: b_h :: σ_r :: p_pv, m, right), p :: b_r :: b_h)
42:                        if (r_1 == unsat ∧ r_2 == unsat) then
43:                            return WeakInf(while (b) (b_v) (b_r :: b_h) {s_b}, σ, m, c, b_cv, b'_cr, b_pv, b_pr)
44:                        else return WeakInf(while (b) (b_v) (b_r) {s_b}, σ, m, c, b_cv, b'_cr, b_pv, b_pr)
45:                        end if
46:                    end if
47:            end match
48:    end match
49: end function
```

Figure 5-5: Weak Inference Algorithm

- An SMT result $r \in R$.
- A set of false conjuncts from the loop invariant b_v.
- A set of false conjuncts from the relational loop invariant b_r.

Strong Inference. Figure 5-4 presents the strong inference algorithm. Before checking a loop, Leto assembles the candidate invariants

$$b_v \equiv p_v$$
$$b_r \equiv p_r \wedge \left(\bigwedge_{x \in \text{vars}} x_r = x_o \right)$$

where p_v is the invariant for the immediate parent loop or function, p_r is the relational invariant for the immediate parent loop or function, and vars is the set of program variables currently in scope.

Leto replaces the programmer provided invariants in the loop with b_v and b_r and invokes the INF function. It also provides the INF function with the programmer provided invariant as b_{pv} and the programmer provided relational invariant as b_{pr}. Leto uses these invariants as assumptions during the inference process. The algorithm has a different behavior for each of the verification modes:

- **Lockstep.** The beginning of the lockstep algorithm (Lines 4 through 8) is similar to the lockstep case for while loop verification, but I've replaced all invocations of Verify with invocations of Check and all applications of Ψ with applications of Ψ'.

  The algorithm proceeds in three possible ways based on the results of the Check function:

  - If any of the three Check results is *unknown*, then Leto falls back on its weak inference algorithm (Line 10).
  - If any of the three Check results is *sat*, the INF function recurses with false conjuncts removed from b_v and b_r (Lines 12 through 14).
  - If all three of the Check results are *unsat*, then the algorithm has converged on a set of invariants and returns (b_v, b_r) (Line 16).

- **Left.** In left mode Leto does no invariant inference due to the fact that Leto does not verify loop invariants in left mode.
- **Right.** The beginning of the right algorithm (Lines 19 through 21) is similar to the right case for while loop verification, but I've replaced all invocations of Verify with invocations of Check and all applications of Ψ with applications of Ψ'.

  The algorithm proceeds in three possible ways based on the results of the Check function:

  - If any of the two Check results is *unknown*, then Leto falls back on its weak inference algorithm (Line 23).
  - If any of the two Check results is *sat*, the INF function recurses with false conjuncts removed from b_v and b_r (Lines 25 through 27).
  - If both of the Check results are *unsat*, then the algorithm has converged on a set a invariants and returns (b_v, b_r) (Line 29).

Weak Inference. Figure 5-5 presents the weak inference algorithm. Leto falls back on this algorithm when any call to the SMT solver returns *unknown*. While the strong inference algorithm iteratively prunes a set of candidate invariants, the weak inference algorithm builds up a set of invariants one at a time from a set of candidates. This is inherently weaker than the strong inference algorithm as it cannot always infer invariants that depend on other invariants. The WEAKINF function takes these candidates as parameters (b_{cv} for standard invariants and b_{cr} for relational invariants) in addition to the loop to perform inference over and the programmer provided invariants for that loop.

The weak inference algorithm operates in three stages:

- **Standard invariant inference (Lines 4 through 23).** Leto adds the head of the standard candidate invariant list (b_h) to the loop invariant then proceeds differently depending on the verification mode:

  - **Lockstep.** The beginning of the lockstep algorithm (Lines 5 through 10) is similar to the lockstep case for while loop verification, but I've replaced all invocations of

Verify with invocations of Check and all applications of Ψ with applications of Ψ' and added the head of the candidate invariant list at each step.

The algorithm proceeds in two possible ways based on the results of the Check function:

  * If all three of the Check results are *unsat*, then the algorithm recurses with b_h appended to b_v and the tail of b_{cv} as the candidate invariant list (Line 12).
  * If any of the three Check results are not *unsat*, then the algorithm discards the candidate invariant and recurses (Line 13).

- **Left.** In left mode Leto does no invariant inference due to the fact that Leto does not verify loop invariants in left mode.
- **Right.** The beginning of the right algorithm (Lines 16 through 18) is similar to the right case for while loop verification, but I've replaced all invocations of Verify with invocations of Check and all applications of Ψ with applications of Ψ' and added the head of the candidate invariant list at each step.

  The algorithm proceeds in two possible ways based on the results of the Check function:

  * If both of the Check results are *unsat*, then the algorithm recurses with b_h appended to b_v and the tail of b_{cv} as the candidate invariant list (Line 20).
  * If any of the two Check results are not *unsat*, then the algorithm discards the candidate invariant and recurses (Line 21).

• **Relational invariant inference (Lines 28 through 46).** After exhausting the standard candidate invariant list, Leto iterates through the relational candidate invariant list. This process is identical to the previous stage but uses b_{cr} in place of b_{cv}.

• **Base case (Line 26).** When no candidate invariants remain, WEAKINF returns the pair of invariants (b_v, b_r).

5.2 Implementation

Leto generates constraints to be solved by Microsoft's Z3 SMT solver. My system makes use of Z3's real, int, and bool types as well as uninterpreted functions for matrices. As such, my system does not necessarily generate a set of constraints for which Z3 is complete. The practical impact of this design is that it is possible for Z3 to be unable to verify valid constraints. Leto provides support for mapping matrices of statically known size to Z3's nlsat solver, which is complete. However, even without this technique, I have been able to successfully verify critical fault tolerance properties for several applications as presented in the following chapters.

Chapter 6

Case Studies

I present four benchmarks that I implemented and verified using Leto and include the results in Figure 6-1. For each benchmark I present the number of lines of code (not including comments or empty lines), the number of annotations I added manually, the number of invariants Leto inferred, the time it took each benchmark to run, the maximum memory usage during verification, and the number of constraints generated.

6.1 Benchmarks and Properties

Jacobi Iterative Method. I verify the Jacobi benchmark as presented in Chapter 2 under the single event upset (SEU) error model I presented in Figure 1-1. Specifically, I verify that the impact of errors on the intermediate solution vector is bounded.

Self-Correcting Connected-Components (SC-CC). SC-CC is an iterative algorithm for computing the connected subgraphs in a graph where each iteration consists of a faulty

Benchmark	LOC	Manual Annotations	Invariants Inferred	Time (s)	Memory Usage (kbytes)	Constraints Generated
Jacobi	57	25	29	9.79	32048	4948
SS-CG	163	26	32	11.51	36696	7554
SS-SD	59	12	0	0.62	25552	418
SC-CC	92	33	43	573.18	95500	10258

Figure 6-1: Benchmark Verification Effort and Runtime Characteristics

initial computation step followed by a correction step [55]. I verify that after each iteration all program state variables are equal across both executions. That is, I verify that all errors are detected and corrected. I verify this property under an error model that allows for faulty writes in the storage vector for the next iteration so long as the errors are large enough to trigger the detector, in contrast the SEU model (Figure 1-1) which places an upper bound on error magnitude. I allow an unbounded number of these errors. I present this error model in Figure 6-2. Additionally, I present a more detailed analysis of SC-CC in Chapter 7.

Self-Stabilizing Conjugate Gradient Descent (SS-CG). SS-CG is an iterative linear system of equations solver that employs a periodic, reliable correction step to repair state variables [54]. I verify under the SEU error model from Figure 1-1 that errors are sufficiently small such that the algorithm does not diverge, and that the correction step can be performed reliably using instruction-level duplication, or dual modular redundancy (DMR). I present a more detailed analysis of SS-CG in Chapter 8.

Self-Stabilizing Steepest Descent (SS-SD). SS-SD is another iterative linear system of equations solver that employs a periodic, reliable correction step [54]. I verify only the correction step, and show that DMR can be used to perform the reliable step under the SEU error model presented in Figure 1-1. I present a more detailed analysis of SS-SD in Chapter 9.

6.2 Verification Effort

Columns 3 and 4 of Figure 6-1 detail the annotation overhead Leto imposes on the programmer. For each benchmark, I present the number of manual annotations, and the number of invariants Leto was able to infer. Manual annotations include loop invariants, assertions, and function requirements. I consider each conjunct a separate annotation when counting inferred invariants and manual annotations.

I significantly reduce the number of invariants I must provide using inference in all but one benchmark. In half of the cases I infer more invariants than I provide. The uninferred invariants fall into two categories:

```
1 const int min_error = ...;
2 @region(unreliable) write(x1, x2) ensures (x1 == x2);
3 @region(unreliable) write(x1, x2) ensures (x2 + min_error < x1);
```

Figure 6-2: Large Errors Execution Model

- **Unsupported Invariants.** Invariants that are not of the form `eq(x)` or are not present in a parent loop or function can not be inferred as Leto does not add them to the set of candidate loop invariants. Expanding the set of candidate loop invariants requires additional heuristics that generate probable loop invariants from a template.

- **Dependant Invariants.** When Leto falls back on weak inference, which is not uncommon due to the fact that many candidate invariants require incomplete Z3 solvers to solve, invariants that depend on the presence of other inferred invariants become order dependant. That is, inference becomes sensitive to the order in which candidate invariants are tested. Leto does not expose a way to reorder these candidate invariants and as such the default ordering (which largely depends on the C++ string hash function) sometimes results in Leto rejecting valid candidates.

I infer no invariants for SS-SD as the inference process very quickly runs out of memory on my machine and therefore I must disable inference on all loops in that benchmark. I believe this issue could be resolved by monitoring the memory usage of the Z3 subprocess, killing the process if it consumes too much, and falling back on my weak inference algorithm.

6.3 Runtime Characteristics

Columns 5 through 7 of Figure 6-1 present the runtime performance characters of the Leto C++ implementation. For each benchmark I present the time it took to run in seconds, the maximum memory usage in kilobytes, and the number of constraints generated for use with Z3. I ran my experiments on an Intel i5-5200U CPU clocked at 2.20GHz with 8 GB of RAM.

```
1  bool stuck = false;
2  bool unstuck = false;
3
4  operator*(x1, x2) when (!stuck)
5      ensures (result == x1 * x2);
6  operator*(x1, x2) when (!unstuck)
7      modifies (stuck) ensures (stuck);
8  operator*(x1, x2) when (stuck && !unstuck)
9      modifies (stuck, unstuck)
10     ensures (result == x1 * x2)
11     ensures unstuck && !stuck;
```

Figure 6-3: Multicycle Error Execution Model

6.4 Execution Models

Leto supports first class, developer provided execution models with arbitrary semantics. However, I have focused exclusively on a single model until this point. Therefore, as an illustration of the expressitivty of Leto's execution model language, I present two additional execution models: one with large errors (Figure 6-2) and one with multicycle fault semantics (Figure 6-3).

Large Errors. I present the large errors execution model in Figure 6-2. Both operator specifications cover variables stored in the `unreliable` memory region. Line 2 specifies a reliable write operator while Line 3 specifies a faulty write operator. The faulty write operator allows for errors so long as they are larger than `min_error`. Developers may use this model as an approximation for rowhammer attack errors. I explore this possibility in Chapter 7.

Multicycle Errors. A multicycle error is an error state in which multiple consecutive instructions experience errors. Huang and Wen [22] have demonstrated that these errors make up a significant percentage of all soft errors. In Figure 6-3 I demonstrate a single multicycle error and track the state of this error through the use of model variables `stuck` and `unstuck`.

Line 4 describes a reliable multiplication implementation that the model may use when `!stuck`. This property holds both before, and after a multicycle error.

Line 6 encodes an operator that the model may use during, or to start a multicycle error.

The model may substitue this operator so long as a multicycle error has not occurred and resolved before the current instruction (`!unstuck`). In other words, the model may use it only to start a multicycle error for the first time or to continue an ongoing mutlicycle error. The operator sets `stuck` and puts no restriction on `result`, therefore allowing unbounded errors.

The model may use the operator on Line 8 only to mark the end of an ongoing multicycle error (`stuck && !unstuck`). It sets `unstuck`, `!stuck`, and returns a reliably computed product of `x1` and `x2`. After substituting this operator the model may only use the first operator (Line 4) as it is the only one that does not require `!unstuck`.

Together, these operators ensure that at some point the system may be stuck experiencing faults on all instructions, but at a later point it may unstick after which all execution is reliable. A full multicycle error specification would define the other operator types similarly, using the same two model variables.

Chapter 7

Self-correcting Connected Components

Figure 7-2 presents an implementation of self-correcting connected components (SC-CC) [55], an iterative algorithm that computes the connected components of an input graph. A connected component is a subgraph in which every pair of vertices in the subgraph is connected through some path, but no vertex is connected to another vertex that is not also in the subgraph.

The standard connected components algorithm begins by constructing a vector CC^0 and initializing this vector such that $\forall v.\ CC^0[v] = v$. Then, on iteration i for each node v the algorithm looks up the value of each of v's neighbors in CC^{i-1} and sets $CC^i[v]$ to the minimum of its neighbors and $CC^{i-1}[v]$. In other words,

$$CC^i[v] = \min_{j \in \mathcal{N}(v)} CC^{i-1}[j] \tag{7.1}$$

where $\mathcal{N}(v)$ is the union of v and the neighbors of node v. The algorithm iterates this process until no elements in CC are updated at which point it has converged.

Self-correcting connected components adds an additional step of checking CC^i after each iteration to verify that it is valid and has not been corrupted by memory errors. If an error is detected at $CC^i[v]$, the computation for node v is repeated with reliably backed storage.

In my implementation I allow errors when writing CC^i so long as the errors are sufficiently

```
1  const real max_N = ..;
2
3  property_r vec_bound(matrix<uint> V, uint i) :
4      forall(uint j)((0 <= j < i<o>) -> (V<o>[j] <= j));
5
6  property_r large_error_r(matrix<uint> V, uint i) :
7    forall(uint j)((0 <= fi < i<r>) ->
8        (V<r>[j] == V<o>[j] || j < V<r>[j]));
9
10 property_r outer_spec(uint to, uint N, matrix<uint> next_CC,
11                       matrix<uint> CC, matrix<uint> adj) :
12   forall(uint fi)((0 <= fi < to) ->
13       (forall(uint fj)((0 <= fj < N && adj[fi][fj] == 1) ->
14           next_CC[fi] <= CC[fj]) &&
15        next_CC[fi] <= CC[fi] &&
16        (exists(uint ej)(next_CC[fi] == CC[ej] && 0 <= ej < N &&
17             adj[fi][ej] == 1) ||
18         next_CC[fi] == CC[fi])));
19
20 property_r inner_spec(uint to, uint v, uint N, matrix<uint> next_CC,
21                       matrix<uint> CC, matrix<uint> adj) :
22   forall(uint fi)((0 <= fi < to && adj[v][fi] == 1) ->
23       next_CC[v] <= CC[fi]) &&
24   next_CC[v] <= CC[v] &&
25   (exists(uint ei)(next_CC[v] == CC[ei] && 0 <= ei < N &&
26        adj[v][ei] == 1) ||
27    next_CC[v] == CC[v]);
```

Figure 7-1: Constant and Properties for Self-Correcting Connected Components

```
1  matrix<uint> cc(uint N, matrix<uint> adj(N, N))
2      requires N < max_N
3      requires_r eq(adj)  {
4    matrix<uint> CC(N);
5    @region(unreliable) matrix<uint> next_CC(N);
6
7    for (uint v = 0; v < N; ++v) invariant_r vec_bound(CC, v) {CC[v] = v;}
8    uint N_s = N;
9
10   @noinf while (0 < N_s)
11       invariant N < max_N
12       invariant_r vec_bound(CC, N)
13       invariant_r eq(N) && eq(adj) && eq(N_s) && eq(CC)  {
14
15     next_CC = CC;
16     N_s = 0;
17
18     for (uint v = 0; v < N; ++v)
19         invariant_r vec_bound(next_CC, N)
20         invariant_r large_error_r(next_CC, N)
21         invariant_r forall(uint fi)((v<o> <= fi < N<o>) -> next_CC<o>[fi] == CC<o>[fi])
22         invariant_r outer_spec(v<o>, N<o>, next_CC<o>, CC<o>, adj<o>) {
23
24       for (uint j = 0; j < N; ++j)
25           invariant v < N && N < max_N
26           invariant_r forall(uint fi)((v<o> < fi < N<o>) -> next_CC<o>[fi] == CC<o>[fi])
27           invariant_r inner_spec(j<o>, v<o>, next_CC<o>, CC<o>, adj<o>)
28           invariant eq(j) {
29         if (CC[j] < next_CC[v] && next_CC[v] <= v && adj[v][j] == 1) {
30           next_CC[v] = CC[j];
31         }
32       }
33     }
34
35     matrix<uint> corrected_next_CC(N);
36     for (uint v = 0; v < N; ++v)
37         invariant_r outer_spec(v<r>, N<r>,
38             corrected_next_CC<r>, CC<r>, adj<r>)
39         invariant_r eq(v) && eq(N_s)
40         invariant_r forall(uint fi)(((0 <= fi < v<r>) ->
41             (corrected_next_CC<r>[fi] == corrected_next_CC<o>[fi])))
42         invariant_r vec_bound(next_CC, N)
43         invariant_r large_error_r(next_CC, N)
44         invariant_r outer_spec(N<o>, N<o>, next_CC<o>, CC<o>, adj<o>) {
45       corrected_next_CC[v] = next_CC[v];
46       if (v < corrected_next_CC[v]) {
47         corrected_next_CC[v] = CC[v];
48         for (uint j = 0; j < N; ++j)
49             invariant v < N && v < next_CC[v]
50             invariant_r inner_spec(j<r> , v<r>, corrected_next_CC<r> CC<r>, adj<r>) {
51           if (CC[j] < corrected_next_CC[v] && adj[v][j] == 1) {
52             corrected_next_CC[v] = CC[j];
53           }
54         }
55       }
56       if (corrected_next_CC[v] < CC[v]) {++N_s;}
57     }
58     CC = corrected_next_CC;
59   }
60   return CC;
61 }
```

Figure 7-2: Self-Correcting Connected Components

large. Therefore, I consider CC^i to be valid if $\forall v.\ 0 \leq CC^i[v] < |V|$ and in all other cases I correct the invalid positions. When this property holds, I can be sure that after each iteration CC<o> == CC<r>, even though intermediate values may differ during faulty execution.

The original SC-CC algorithm described by Sao et al. contains an additional data structure P* and permits a larger class of errors than my implementation does. However, this flexibility comes at a cost: the original algorithm is not guaranteed to converge. As such, I modified the algorithm to prove strong convergence properties not offered by the original.

Self Correction. SC-CC is *self correcting.* This means that given some valid state, SC-CC can correct errors encountered during each iteration. In this case, if an error occurs at iteration i, CC^i can be corrected using data from CC^{i-1}. Therefore, CC for the previous iteration must always be stored correctly. This is weaker than self-stabilizing algorithms which may correct themselves from any state and do not rely on certain state elements remaining uncorrupted.

7.1 SC-CC Implementation

The overall structure of the SC-CC implementation is as follows:

- **Initialization.** The cc function takes a description of a graph in the form of an adjacency matrix (adj). It then declares and initializes CC, which holds the result of the previous iteration. It also declares next_CC, which holds the result of the current iteration, in the unreliable memory region.

- **Outer while loop (Line 10)** The outer while loop computes the next iteration of CC. It converges when the algorithm makes no changes to CC over the course of a single iteration.

- **Faulty step (Line 18)** The faulty step computes Equation 7.1 element-wise over next_CC. The inner loop allows errors during writes to next_CC, which will be corrected in the correction step.

- **Correction step (Line 36)** The correction step computes `corrected_next_CC` by detecting errors in `next_CC`. If no error has occurred at index `v`, the implementation reliably copies `next_CC[v]` to `corrected_next_CC[v]`. Otherwise, the implementation reliably computes `corrected_next_CC[v]` using Equation 7.1. After computing `corrected_next_CC`, the implementation sets `CC` equal to `corrected_next_CC` and begins the next iteration.

Constants and Properties. My SC-CC implementation uses the following constants and properties, found in Figure 7-1:

- **`max_N` (Line 1).** This constant bounds the maximum number of nodes an input graph may contain.

- **`vec_bound` (Line 3).** This property takes a vector `V` and an index `i` and stipulates that $\forall\ \texttt{j} < \texttt{i<o>}.\ \texttt{V<o>[j]} \leq \texttt{j}$.

- **`large_error_r` (Line 6).** This property takes a vector `V` and an index `i` and asserts that $\forall\ \texttt{j} < \texttt{i<r>}.\ \texttt{V<r>[j]} = \texttt{V<o>[j]} \lor \texttt{j} < \texttt{V<r>[j]}$.

- **`outer_spec` (Line 10).** This property takes an index `to`, a size `N`, a vector `next_CC`, a vector `CC`, and an adjacency matrix `adj`. It ensures that every element of `next_CC` from index 0 to index `to` (exclusive) satisfies Equation 7.1.

- **`inner_spec` (Line 20).** This property takes an index `to`, an index `v`, a size `N`, a vector `next_CC`, a vector `CC`, and an adjacency matrix `adj`. It ensures that `next_CC[v]` satisfies Equation 7.1 up to the neighbor at index `to`. That is,

$$\texttt{next_CC[v]} = \min_{j \in \mathcal{N}(\texttt{v, to})} \texttt{CC[j]}$$

  where $\mathcal{N}(\texttt{v, to})$ is the union of `v` and the neighbors of node `v` up to (but not including) nodes with the id `to`.

Error Model. I evaluate SC-CC under the large errors fault model presented in Figure 6-2. This model provides two implementations for the write operator in the memory region

unreliable. The first implementation is fully reliable while the second allows for errors so long as they are larger than the programmer specified constant min_error.

I use this fault model that allows only faulty writes because it captures all possible rowhammer attacks over high order bits in the elements of next_CC. A rowhammer attack allows an attacker to selectively flip bits in DRAM by issuing frequent reads on DRAM rows surrounding the row under attack [26]. This drains capacitors in the attacked row and therefore permanently flips bits in that row. As empty capacitors may indicate a 0 or 1 depending on location on the chip, this attack does not always flip bits from 1 to 0. Although researchers have devised protections to address rowhammer attacks [25, 26], the JDEC Solid State Technology Association did not include these protections in the DDR4 standard [2] and Mark Lanteigne has demonstrated rowhammer attacks on some DDR4 memory [29].

Given that some regions of memory may be more vulnerable to rowhammer attacks than others [26], I place next_CC in an unreliable region prone to attacks (Line 5) and all other variables in reliable memory.

Although the semantics of Leto over this fault model allow errors only at the site of memory writes, I demonstrate that this captures all possible rowhammer attacks over high order bits in next_CC by breaking my implementation into regions in which rowhammer errors may occur and demonstrating that errors in those regions are equivalent to errors the fault model may inject in conjunction with the set of invariants I have provided.

- **Lines 4-10.** This region consists of everything up to the outer while loop. This region does not modify next_CC in any way and the outer while loop puts no constraints on next_CC. Therefore, Leto permits unbounded errors to next_CC in this region.

- **Lines 15-18.** This region consists of the initialization code between the top of the outer while loop and the middle faulty for loop. Leto captures rowhammer errors in this region through the constraints generated from the full vector copy on Line 15. That is, rowhammer errors encountered in high order bits after the vector copy exhibit behavior captured by the constraints generated during the vector copy as these errors are at least as large as min_error and are permanent.

- **Lines 24-30.** This region includes everything in the middle faulty loop. I break

rowhammer errors encountered in this region into three categories based on whether they modify elements before, during, or after they are updated:

- **Before update.** Errors that occur before the conditional on Line 29 are preserved as these errors will cause the conjuct `next_CC[v] <= v` to evaluate to false. This case is identical to the case where an error during the copy on Line 15 causes an error at this element.
- **During update.** Errors that occur after checking the condition but before updating `next_CC` on Line 30 are instantly clobbered by the update and can therefore be ignored.
- **After update.** Errors that occur after the faulty update are equivalent to errors incurred during the faulty write step on Line 30.

- **Line 35.** Rowhammer errors that occur after exiting the faulty outer loop but before entering the outer correction loop are consistent with the `large_error_r(next_CC,N)` invariants on Lines 20 and 43 and Leto therefore considers the vectors that would result from these errors as valid.

- **Lines 46-58.** This region includes the outer correction loop. Errors in `next_CC` can occur before, or after the copy on Line 45. If a rowhammer error occurs in an element before that element is copied into `corrected_next_CC`, it will be detected by the if statement on the following line, thus triggering the correction step. If the error to that element occurs after it has already been copied it has no impact as the implementation does not reference that element again until the next outer while iteration at which point it is overwritten. Additionally, the `large_error_r(next_CC, N)` invariant includes the class of errors experienced through high order bit rowhammer bit flips, so there are no unforeseen consequences spanning multiple loop iterations.

- **Line 58 and Line 60** These two lines do not depend on `next_CC` so any rowhammer errors the implementation experiences here have no impact on the program. Additionally, the implementation overwrites any errors on `next_CC` in the next loop iteration. Finally,

the outer while loop places no restrictions on next_CC so Leto finds any errors over the vector to be valid.

7.2 Specification

I use Leto's specification abilities to verify the error detection, self-correction, and convergence properties of SC-CC.

Perfect Error Detection. To verify that this SC-CC implementation detects all errors, the developer must verify that N < max_N, where N is the number of nodes in the input graph and max_N is a programmer specified value bounding the maximum graph size a calling function may provide. The developer must also ensure that max_N is less than the magnitude of the largest error they expect to encounter. This property ensures that any error will be larger than N, which in turn ensures that all errors are detected as errors result in invalid values that are easily detected. I specify this property as a unary prerequisite to calling the cc function on Line 2. I also specify this property as unary invariants that must hold before and after each loop iteration on Lines 11 and 25. Although not explicitly stated, Leto infers this invariant for the loop on Line 18.

Error detection also requires that impact of errors on next_CC<r> is large enough to be detected. Specifically, it is necessary that

$$\forall i.\ \texttt{next_CC<r>}[i] > i \lor \texttt{next_CC<r>} == \texttt{next_CC<o>}.$$

This property ensures that for every element e of next_CC<r> at index i, e is in one of two states:

- **Detectable Error.** When $e > i$, e violates the property that every element of CC does not increase from its initialization value. This error is trivially detected by the check in the conditional on Line 46 of the correction step.

- **Equality.** When $e = \texttt{next_CC<o>}[i]$, e is correctly computed.

Note that e can not be in a state where it contains an undetectable error. Loop invariants on Lines 20 and 43 (large_error_r(next_CC, N)) ensure this property.

The final property I require to ensure that my implementation has perfect error detection is that adj<o> == adj<r>. This property asserts that the graphs both executions operate over are equivalent, and unable to experience errors. I enforce this property by specifying it as a requirement to call the cc function on Line 3. On all loops I either explicitly specify eq(adj), or Leto infers it.

Self-Correction. To verify that my SC-CC implementation is self correcting, I first verify that every element of next_CC<o> satisfies Equation 7.1. I capture this with the outer_spec property application on Line 22 and the inner_spec property application on Line 27. Both of these properties capture the semantics of the min operator. After exiting inner faulty loop (j == N), inner_spec(j<o>, v<o>, next_CC<o>, CC<o>, adj<o>) is equivalent to Equation 7.1 at index v. Similarly, after exiting the outer loop (v == N), the application outer_spec(v<o>, N<o>, next_CC<o>, CC<o>, adj<o>) is equivalent to Equation 7.1 at all indices, or

$$\forall v.\ CC^{i}[v] = \min_{j \in \mathcal{N}(v)} CC^{i-1}[j].$$

The same two properties (outer_spec and inner_spec) are applied in the same fashion with corrected_next_CC<r> in place of next_CC<o> (Lines 37 and 50 respectively) to specify that corrected_next_CC<r> also satisfies Equation 7.1 at all indices.

I pass the specification of next_CC<o> into the correction loop on Line 44 where Leto combines it with the specification of corrected_next_CC<r> to prove that

∀i < v<r>. corrected_next_CC<r>[i] == corrected_next_CC<o>[i],

stated on Line 40.

Finally, with the assignment CC = corrected_next_CC Leto can verify the outer loop invariant eq(CC) (Line 13) thus proving that my implementation is self correcting.

Convergence Equality. Given that my SC-CC implementation is self correcting and detects all errors, it is trivial to see that both executions converge in the same number of iterations. That is, the outer while loop (Line 10) must run in lockstep. To demonstrate this fact to Leto I use the eq(N_s) invariant on Lines 13 and 39.

N_s itself is updated in two places:

- **Line 16**. I set N_s to 0 at the top of the outer loop. As N_s is stored in reliable memory, it is clear that N_s<o> == N_s<r>.

- **Line 56**. I increment N_s at this location if corrected_next_CC[v] < CC[v]. From the surrounding loop invariants Leto knows that

  ```
  corrected_next_cc<o>[v<o>] == corrected_next_CC<r>[v<r>]
  ```

  and

  ```
  CC<o>[v<o>] == CC<r>[v<r>]
  ```

  so the if statement must execute in lockstep. Therefore, if N_s was equal across both executions before the if statement, then it is equal afterwards as the increment to N_s is performed reliably.

Leto realizes that N_s is equal across both executions after both assignments and therefore eq(N_s) must hold in both loop invariants. This forces the outer while loop into a lockstep execution and proves that convergence time is equal under relaxed and reliable execution semantics.

7.3 Verification Approach

I next demonstrate how the developer works with Leto to verify that the implementation meets these specifications.

Precondition. The verification algorithm begins with the preconditions on cc. The precondition stipulates that N be less than max_N and that the graph be equal across both

executions (`eq(adj)`). Leto adds these preconditions as assumptions to its context.

Initialization On Line 7 I initialize the `CC` vector such that $\forall$`v`. `CC[v]` $=$ `v`. The loop invariant verifies that all elements with index `v` in `CC` are between 0 and `v`. This property is critical in the detection of errors and must hold for `CC` after each iteration of the connected components algorithm.

Outer Loop The loop on Line 10 runs the iterative portion of the algorithm. The loop enforces the critical invariants `vec_bound(CC, N)`, which ensures that $\forall$`v`. `CC[v]` $\leq$ `v`, and `eq(CC)`. It also contains the invariant `eq(N_s)`, which ensures that the loop runs in lockstep. Lastly, it enforces the invariants `eq(N)` and `eq(adj)` which ensure that the input graph is identical across both executions.

The loop also sports the `@noinf` annotation, which disables inference over this loop. I disable inference on this loop because the inference algorithm's time complexity is exponential in the depth of nested loops.

Faulty Middle Loop. Verification then proceeds to the faulty middle loop on Line 18. This loop contains the following invariants:

- **`vec_bound(next_CC, N)`.** This invariant enforces that elements of `next_CC<o>` are bounded by their respective indices. This fact is important to pass on to the correction step as it implies that the original execution never runs the inner correction loop.

- **`large_error_r(next_CC, N)`.** This invariant enforces that errors to `next_CC<r>` are large enough to be detected. It enables the implementation to detect and correct all errors during the correction step.

- **`forall(uint fi)((v<o> <= fi < N<o>) -> next_CC<o>[fi] == CC<o>[fi])`.** This invariant states that elements in `next_CC` that the implementation hasn't yet updated are still equal to `CC`. This is necessary as it communicates to Leto that if an element is not updated on this iteration, then it is already the minimum of its neighbors.

- **outer_spec(v<o>, N<o>, next_CC<o>, CC<o>, adj<o>).** This invariant specifies the contents of next_CC<o>. I use it to pass information about next_CC<o> on to the correction step.

Faulty Inner Loop. Verification then continues to the loop on Line 24, where it encounters the following invariants:

- **v < N.** This invariant bounds v so that Leto knows that the vector accesses within this loop are in bounds.

- **N < max_N.** This invariant bounds the maximum size of the graph so that Leto knows that errors are larger than the maximum graph size and are therefore detectable.

- **forall(uint fi)((v<o> < fi < N<o>) -> next_CC<o>[fi] == CC<o>[fi]).** This invariant serves to pass on that although the implementation may have altered next_CC<o>[v<o>], the other elements that Leto previously knew were equal to CC<o> in the middle faulty loop are still equal to CC<o>.

- **inner_spec(j<o>, v<o>, next_CC<o>, CC<o>, adj<o>).** This invariant specifies the contents of next_CC<o>[v<o>]. It serves only to pass this information on to the outer loop so Leto may verify the outer_spec invariant in that loop.

- **eq(j).** This invariant states that j<o> == j<r>. It informs Leto that this loop must run in lockstep. Leto can not infer this property because it falls back to weak inference on this loop, eq(j) is a candidate invariant before eq(N), and eq(j) does not hold if N<r> != N<o>.

Correction Middle Loop. Verification next proceeds to the loop on Line 36. This loop contains the following invariants:

- **outer_spec(v<r>, N<r>, corrected_next_CC<r>, CC<r>, adj<r>).** This invariant specifies the contents of corrected_next_CC<o>. Leto combines it with the other outer_spec invariant to verify corrected_next_CC<o> == corrected_next_CC<r> after exiting the loop.

- **eq(v).** This invariant states that v<o> == v<r>. It informs Leto that this loop must run in lockstep. Leto can not infer this property because it falls back to weak inference, is a candidate invariant before eq(N), and does not hold if N<r> != N<o>.

- **eq(N_s).** This invariant states that N_s<o> == N_s<r>. Although it is a candidate for inference, Leto falls back on weak inference for this loop. As it depends on proving eq(corrected_next_CC), which itself is a complicated invariant depending on many other invariants, weak inference is not capable of inferring this invariant. Leto uses this invariant to relay the information that N_s is equal across both executions to the outer loop.

- **forall(uint fi)(((0 <= fi < v<r>) -> (corrected_next_CC<r>[fi] == corrected_next_CC<o>[fi]))).** This invariant states that elements in corrected_next_CC that the implementation has updated are equal across both executions. This allows Leto to prove eq(CC) at the end of the outer while loop.

- **vec_bound(next_CC, N).** This invariant enforces that elements of next_CC<o> are bounded by their respective indices. This implies that the original execution never runs the inner correction loop. Therefore, Leto prunes any paths in which the original execution runs through the inner loop.

- **large_error_r(next_CC, N).** This invariant enforces that errors to next_CC<r> are large enough to be detected. It enables the implementation to detect and correct all errors during the course of this loop.

- **outer_spec(N<o>, N<o>, next_CC<o>, CC<o>, adj<o>).** This invariant passes in the specification for next_CC<o> from the faulty loop. Leto combines it with specifications over corrected_next_CC<r> to prove that corrected_next_CC<o>[v<o>] is equivalent to corrected_next_CC<r>[v<r>] at the end of the current loop iteration.

Correction Inner Loop. Verification then continues to the loop on Line 48 which contains three developer specified invariants:

- **v < N.** This invariant bounds v so that Leto knows that the vector accesses within this loop are in bounds.

- **v < next_CC[v].** This invariant ensures that this loop only runs in instances where next_CC[v] has experienced an error. The conditional that contains this loop implies this invariant.

- **inner_spec(j<r> , v<r>, corrected_next_CC<r> CC<r>, adj<r>).** This invariant specifies the contents of corrected_next_CC<r>[v<r>]. It serves only to pass this information on to the outer loop so that Leto may verify the outer_spec in that loop.

Chapter 8

Self-stabilizing Conjugate Gradient Descent

I verify an implementation of self-stabilizing conjugate gradient (SS-CG) [54] and present relevant snippets required for verification below. Conjugate gradient descent is another method for solving linear systems of equations. However, unlike Jacobi, the standard conjugate gradient method is sensitive to errors that may corrupt internal state variables and, therefore, is not naturally self-stabilizing. SS-CG employs a periodic correction step to recalculate appropriate values for internal state variables from the current estimated solution vector `x` and the input matrix `A`.

The standard conjugate gradient descent algorithm computes the next iteration's variables as follows:

$$q_i = Ap_i \tag{8.1}$$

$$\alpha_i = \frac{r_i^T r_i}{p_i^T q} \tag{8.2}$$

$$x_{i+1} = x_i + \alpha p_i \tag{8.3}$$

$$r_{i+1} = r_i - \alpha q_i \tag{8.4}$$

$$\beta = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i} \tag{8.5}$$

$$p_{i+1} = r_{i+1} + p_i \tag{8.6}$$

```
1  property_r dmr_eq(matrix<real> x1, matrix<real> x2, matrix<real> sx) :
2    x1<r> == sx && x2<r> == sx;
3
4  property_r dmr_imp(matrix<real> x1, matrix<real> x2, matrix<real> sx) :
5    (x1<r> == x2<r>) -> (x1<r> == sx);
6
7  matrix<real> r2(N), q2(N);
8  specvar matrix<real> spec_r(N), spec_q(N);
9  r = r2 = spec_r = q = q2 = spec_q = zeros;
10 bool not_run = true;
11
12 @noinf while (not_run || r != r2 || q != q2)
13              invariant_r !model.upset -> (dmr_eq(r, r2, spec_r) &&
14                                           dmr_eq(q, q2, spec_q))
15              invariant_r dmr_imp(r, r2, spec_r)
16              invariant_r dmr_imp(q, q2, spec_q) {
17   not_run = false;
18
19   for (int i = 0; i < N; ++i) {
20     for (int j = 0; j < N; ++j) {
21       real tmp = A[i][j] *. x[j];
22       real tmp2 = A[i][j] *. x[j];
23       specvar real spec_tmp = A[i][j] * x[j];
24       r[i] = r[i] +. tmp;
25       r2[i] = r2[i] +. tmp2;
26       spec_r[i] = spec_r[i] + spec_tmp;
27
28       tmp = A[i][j] *. p[j];
29       tmp2 = A[i][j] *. p[j];
30       spec_tmp = A[i][j] * p[j];
31       q[i] = q[i] +. tmp;
32       q2[i] = q2[i] +. tmp2;
33       spec_q[i] = spec_q[i] + spec_tmp;
34     }
35   }
36 }
37
38 assert_r (!outer_while[model.upset] -> (r<r> == spec_r));
39 assert_r (!outer_while[model.upset] -> (q<r> == spec_q));
```

Figure 8-1: SS-CG Correction Step

```
1  const real M = ...;
2
3  property_r sqr_lt(matrix<real> v, int i) :
4    ((v<r>[i<r>] - v<o>[i<o>]) * (v<r>[i<r>] - v<o>[i<o>])) < M;
5
6  for (int i = 0; i < N; ++i)
7      invariant 0 <= i <= N {
8
9    q[i] = 0;
10   @label(inner_err)
11   for (int j = 0; j < N; ++j)
12       invariant 0 <= j <= N && 0 <= i < N
13       invariant_r (!model.upset && eq(p)) -> q<r>[i<r>] == q<o>[i<o>] {
14     real tmp = A[i][j] *. p[j];
15     q[i] = q[i] +. tmp;
16
17     assert_r((!inner_err[model.upset] && eq(p)) -> sqr_lt(q, i));
18   }
19 }
```

Figure 8-2: SS-CG Faulty Matrix Vector Product

SS-CG adds a periodic correction step to repair state variables that may have been corrupted by errors. Unlike the previous steps, the repair step must be computed reliably. This repair step computes:

$$r'_i = Ax_i \tag{8.7}$$

$$q_i = Ap_i \tag{8.8}$$

$$r_i = b - r'_i \tag{8.9}$$

$$\alpha_i = \frac{r_i^T p_i}{p_i^T q} \tag{8.10}$$

$$x_{i+1} = x_i + \alpha p_i \tag{8.11}$$

$$r_{i+1} = r_i - \alpha q \tag{8.12}$$

$$\beta = -\frac{r_{i+1}^T q_i}{p_i^T q} \tag{8.13}$$

$$p_{i+1} = r_{i+1} + \beta p_i \tag{8.14}$$

I verify two properties of SS-CG that are necessary for self-stability under the SEU model I present in Figure 1-1:

- **Reliable Correction Step.** I verify that it is possible to correct errors even when the matrix vector products in Equations 8.7 and 8.8 may experience faults. To accomplish this, I use dual modular redundancy (DMR) to duplicate arithmetic instructions and repeat the correction step until the result of both sets of instructions agree with each other.

- **Correctable Errors.** I verify that errors in the matrix vector product from Equation 8.1 are sufficiently small. Specifically, SS-CG requires that if an element of q is corrupted by ϵ, then

$$\epsilon^2 < \max_{i,j}(A[i][j])^2.$$

I present the code snippet for the reliable correction step in Figure 8-1 and the snippet for correctable errors in Figure 8-2. I have included the full implementation in Appendix B.

8.1 Implementation

8.1.1 Reliable Correction Step

The SS-CG correction step I present in Figure 8-1 operates over the following pre-existing variables:

- **A.** `A` is a matrix of coefficients.
- **x.** `x` is a solution vector.
- **r.** `r` holds the residual of the current iteration.
- **p and q.** `p` and `q` are vectors of loop carried state.

The overall structure of the SS-CG correction step is as follows:

- **Initialization (Line 7).** Initialization declares the following variables:
  - **r2 and q2.** The algorithm computes `r2` according to Equation 8.7 and `q2` according to Equation 8.8. It then uses `r2` and `q2` to verify that it has correctly computed `r` and `q` respectively.

- **spec_r and spec_q.** spec_r and spec_q are specification variables that also compute Equation 8.7 and Equation 8.8 respectively. Unlike r, r2, q, and q2, the implementation computes these specification variables reliably.
- **Outer while loop (Line 12).** The outer while loop repeats the correction step until r == r2 and q == q2.
- **Middle for loop (Line 19).** The middle for loop computes Equation 8.7 element-wise for r, r2, and spec_r and Equation 8.8 element-wise for q, q2, and spec_q.
- **Inner for loop (Line 20).** The inner for loop computes the matrix vector products:

$$\texttt{r} = \texttt{A * x}$$

$$\texttt{q} = \texttt{A * p}$$

It computes r2 and spec_r similarly to r, and q2 and spec_q similarly to q. The algorithm permits errors in the computation of r, r2, q, and q2, but not in spec_r or spec_q.

Properties. My SS-CG correction step implementation uses the following properties, found in Figure 8-1:

- **dmr_eq (Line 1).** This property asserts that x1 and x2 are both equal to the specification variable sx.
- **dmr_imp (Line 4).** This property asserts that if x1 and x2 are equal, then x1 is equal to the specification variable sx.

8.1.2 Faulty Matrix Vector Product

The SS-CG faulty matrix vector product I present in Figure 8-2 operates over the following pre-existing variables:

- **A.** A is a matrix of coefficients.

- **p and q.** `p` and `q` are vectors of loop carried state.

The structure of the SS-CG faulty matrix vector product is as follows:

- **Outer for loop (Line 6).** The outer for loop computes Equation 8.1 element-wise over `q`.

- **Inner for loop (Line 11).** The inner for loop computes the unreliable matrix vector product `q` = `A * p`.

Constants and Properties The SS-CG faulty matrix vector product uses the following constants properties, found in Figure 8-2:

- **`M` (Line 1).** `M` represents the maximum square error permissible in a single element of `q`. The developer must set it according to the formula

$$\texttt{M} < \min_{a \in A} \left(\max_{(i,j)} (a[i][j])^2 \right)$$

  where A is the set of `A` input matrices the developer expects to run our implementation over.

- **`sqr_lt` (Line 3).** `sqr_lt` takes a vector `v`, and index `i`, and ensures that the square error of `v[i]` is strictly less than `M`. In other words, it mandates that

$$(\texttt{v<r>[i<r>]} - \texttt{v<o>[i<o>]})^2 < \texttt{M}.$$

8.2 Specification

I use Leto's specification abilities to verify the error correction and small errors properties of SS-CG.

Error Correction Using DMR, the correction step corrects `r` and `q` even in the presence of errors. I enforce this property through the assertions on Lines 38 and 39 of Figure 8-1. As

the algorithm computes `spec_r` and `spec_q` correctly, I know that if `r<r> == spec_r` and `q<r> == spec_q`, then the system has computed `r<r>` and `q<r>` correctly.

The `dmr_imp` and `dmr_eq` property applications in the outer while loop invariants (Lines 13 through 16) pass the information Leto needs to verify this assertion out of the loop. Leto infers these invariants for the inner loops, thus allowing the outer loop to verify that the invariant holds after the modifications the inner loop performs.

Correctable Errors Under SEU, SS-CG requires that if an element of `q` is corrupted by ϵ, then

$$\epsilon^2 < \max_{(i,j)}(A[i][j])^2. \tag{8.15}$$

I enforce this property through the assertion on Line 17 of Figure 8-2. The invariant on Line 13 enforces the complementary invariant that if no error occurred, then `q<r>[i<r>]` is equal to `q<o>[i<o>]`. `eq(p)` guards both of these constraints because we verify this section in isolation without specifying the global properties of `p`. However, if no upset occurred prior to the start of this section then `eq(p)` trivially holds before and throughout as this snippet does not modify `p`.

8.3 Verification Approach

Next, I demonstrate how the developer works with Leto to verify that the implementation meets these specifications.

8.3.1 Correction Step

Outer Loop. Verification begins with the loop on Line 12 of Figure 8-1. This loop contains the following invariants:

- `!model.upset -> (dmr_eq(r, r2, spec_r) && dmr_eq(q, q2, spec_q))`.
  This invariant enforces that in the absence of errors during a loop iteration, `r<r>` is equal to `spec_r` and `q<r>` is equal to `spec_q`. That is, if no errors occur then `r` and `q`

are correct. This fact is important to pass on so that Leto may verify the assertions on Lines 38 and 39.

- **dmr_imp(r, r2, spec_r).** This invariant states that if the duplicated r variables are equal to each other, then r is also equal to spec_r. Combining this with the loop condition, Leto knows that r<r> == spec_r after exiting the loop.

- **dmr_imp(q, q2, spec_q).** This invariant states that if the duplicated q variables are equal to each other, then q is also equal to spec_q. Combining this with the loop condition, Leto knows that q<r> == spec_q after exiting the loop.

Assertions. Verification concludes with the assertions on Lines 38 and 39:

- **!outer_while[model.upset] - > (r<r> == spec_r).** This assertion verifies that if no upset occurred prior to entering the loop containing the outer while loop, then the correction step computed r correctly.

- **!outer_while[model.upset] - > (q<r> == spec_q).** This assertion verifies that if no upset occurred prior to entering the loop containing the outer while loop, then the correction step computed q correctly.

8.3.2 Faulty Matrix Vector Product

Outer Loop Verification begins with the loop on Line 6 of Figure 8-2. This loop contains the invariant 0 <= i <= N, which bounds i to ensure Leto that the loop code contains no out of bounds array accesses.

Inner Loop. Verification then proceeds to the inner loop on Line 11. This loop contains the following invariants:

- **0 <= j <= N && 0 <= i < N.** This invariant bounds i and j to ensure Leto that the loop code does not contain any out of bounds array accesses.

- **(!model.upset && eq(p)) -> q<r>[i<r>] == q<o>[i<o>].** This invariant ensures that if no upset occurred and p<o> == p<r>, then q[i] is equal across both executions.

Assertion. Verification concludes with the assertion on Line 17. This assertion verifies that if no error had occurred prior to the top of the inner loop, and `p<o> == p<r>`, then the square difference between `q<o>` and `q<r>` is less than `M`. This ensures that errors are sufficiently small to be correctable. That is, it ensures that the error in `q` satisfies Equation 8.15.

Chapter 9

Self-Stabilizing Steepest Descent Correction Step

Figure 9-2 presents an implementation of the correction step from self-stabilizing steepest descent (SS-SD) [54]. SS-SD is an iterative algorithm that computes the solution to a linear system of equations. It takes as input a matrix of coefficients `A`, a vector `b` of intercepts, and returns an approximate solution vector `x` such that $\mathtt{A} * \mathtt{x} \approx \mathtt{b}$. On each iteration, steepest descent uses r_i, x_i, and A to compute r_{i+1} and x_{i+1} as follows:

$$q_i = Ar_i \tag{9.1}$$

$$\alpha_i = \frac{r_i^T r_i}{r_i^T q_i} \tag{9.2}$$

$$x_{i+1} = x_i + \alpha_i r_i \tag{9.3}$$

$$r_{i+1} = r_i - \alpha_i q_i \tag{9.4}$$

SS-SD adds a periodic correction step to repair the residual r of any errors it may have incurred. This repair step computes

$$r_i = b - Ax. \tag{9.5}$$

```
1  property_r trans() : outer[model.upset] -> model.upset;
2
3  property_r upset(matrix<real> r, matrix<real> r2, matrix<real> spec_r,
4                   matrix<real> Ax, matrix<real> Ax2,
5                   matrix<real> spec_Ax) :
6    ((!outer[model.upset] && model.upset) ->
7      ((r<r> == spec_r && Ax<r> == spec_Ax) ||
8       (r2<r> == spec_r && Ax2<r> == spec_Ax))) &&
9    ((!model.upset || outer[model.upset]) ->
10     (r<r> == spec_r && r2<r> == spec_r &&
11      Ax<r> == spec_Ax && Ax2<r> == spec_Ax));
12
13 property_r outer(matrix<real> r, matrix<real> spec_r) :
14   (!model.upset || outer[model.upset]) -> r<r> == spec_r)
```

Figure 9-1: SS-SD Properties

Unlike Equation 9.2 through Equation 9.4, SS-SD requires that Equation 9.5 is performed reliably. Therefore, I verify that it is possible to correct errors under the SEU execution model from Figure 1-1 even when the error correction step may experience errors. To accomplish this, I use dual modular redundancy (DMR) to duplicate arithmetic instructions and repeat the correction step until the result of both sets of instructions agree with each other.

9.1 SS-SD Correction Step Implementation

The overall structure of the SS-SD correction step is as follows:

- **Initialization.** The `correct_sd` function takes a matrix of coefficients `A`, a vector of intercepts `b`, and a solution vector `x`. It then declares:
  - **r.** `r` holds the residual that will be returned.
  - **r2.** The function computes `r2` according to Equation 9.5. The algorithm uses `r2` to verify that it has correctly computed `r`.
  - **spec_r.** `spec_r` is a specification variable that also computes Equation 9.5. Unlike `r` and `r2`, the implementation computes `spec_r` correctly.
- **Outer while loop (Line 11).** The outer while loop repeats the correction step until `r == r2`.

```
1  matrix<real> correct_sd(int N, matrix<real> A(N, N),
2                           matrix<real> b(N), matrix<real> x(N)) {
3    matrix<real> zeros(N);
4    @noinf for (int i = 0; i < N; ++i)  { zeros[i] = 0; }
5
6    matrix<real> r(N), r2(N);
7    specvar matrix<real> spec_r(N);
8    spec_r = r;
9    bool run = false;
10
11   @noinf @label(outer) while (run == false || r != r2)
12         invariant_r outer(r, spec_r)
13         invariant_r r<r> == r2<r> -> r<r> == spec_r
14         invariant_r trans() {
15     run = true;
16
17     matrix<real> Ax(N), Ax2(N);
18     specvar matrix<real> spec_Ax(N);
19     Ax = Ax2 = spec_Ax = r = r2 = spec_r = zeros;
20
21     @noinf for (int i = 0; i < N; ++i)
22         invariant_r outer(r, spec_r)
23         invariant_r upset(r, r2, spec_r, Ax, Ax2, spec_Ax)
24         invariant_r trans() {
25
26       @noinf for (int j = 0; j < N; ++j)
27           invariant_r upset(r, r2, spec_r, Ax, Ax2, spec_Ax)
28           invariant_r trans() {
29
30         real tmp = A[i][j] *. x[j];
31         real tmp2 = A[i][j] *. x[j];
32         specvar real spec_tmp = A[i][j] * x[j];
33
34         Ax[i] = Ax[i] +. tmp;
35         Ax2[i] = Ax2[i] +. tmp2;
36         spec_Ax[i] = spec_Ax[i] + tmp;
37       }
38
39       r[i] = b[i] - Ax[i];
40       r2[i] = b[i] - Ax2[i];
41       spec_r[i] = b[i] - spec_Ax[i];
42     }
43   }
44
45   assert_r (r<r> == spec_r);
46
47   return r;
48 }
```

Figure 9-2: SS-SD Correction Step

- **Middle for loop (Line 21).** The middle for loop computes Equation 9.5 element-wise for `r`, `r2`, and `spec_r`.
- **Inner for loop (Line 26).** The inner for loop computes the matrix vector product `Ax` = `A` $*$ `x`. It computes `Ax2` and `spec_Ax` similarly. The algorithm permits errors in the computation of `Ax` and `Ax2`, but not in `spec_Ax`.

Properties. My SS-SD correction step implementation uses the following properties, found in Figure 9-1:

- **`trans` (Line 1).** This property asserts if there was an upset before the top of the outer loop, then the upset flag in the model is set.
- **`upset` (Line 3).** This property consists of two conjuncts:
  - The first conjunct states that if there was no upset before the top of outer loop, but there has been an upset since then, then at least one of the duplicated computations is correct. That is, `r<r> == spec_r && Ax<r> == spec_Ax` or `r2<r> == spec_r && Ax2<r> == spec_Ax`.
  - The second conjunct states that if no upset has occurred, or an upset occurred prior to the top of the outer while loop, then both sets of duplicated instructions are correct.
- **`outer` (Line 13).** This property states that if there has been no upset, or there was an upset prior to the top of the outer loop, then `r` is correct.

9.2 Specification

I use Leto's specification abilities to verify the error correction property of SS-SD's correction step.

Error Correction. Using DMR, the correction step corrects `r` even in the presence of errors. I enforce this property through the assertion on Line 45. As the algorithm computes `spec_r` correctly, I know that if `r<r> == spec_r`, then `r<r>` is the correct residual.

The outer constraint on the outer while loop (Line 12) passes the information needed to verify this assertion out of the loop. In turn, this invariant relies on the outer and upset invariants in the middle loop (Lines 22 and 23), which themselves rely on the upset invariant in the inner loop (Line 27).

9.3 Verification Approach

Next, I demonstrate how the developer works with Leto to verify that the implementation meets these specifications.

Outer Loop. Verification begins with the loop on Line 11. This loop contains the following invariants:

- **outer(r, spec).** This invariant enforces that in the absence of errors during a loop iteration, r<r> == spec_r. This fact is important to pass on so that Leto may verify the assertion on Line 45.
- **r<r> == r2<r> -> r<r> == spec_r.** This invariant states that if the duplicated r variables are equal to each other, then r is also equal to spec_r. Therefore, Leto knows that r<r> == spec_r after exiting the loop.
- **trans().** This invariant passes the semantics of the fault model down into the middle loop so Leto knows that if an error already occurred in an earlier outer loop iteration, then another cannot occur in any inner loops.

Middle Loop. Verification then proceeds to the middle loop on Line 21. This loop contains the following invariants:

- **outer(r, spec).** This invariant enforces that in the absence of errors during an outer loop iteration, r<r> == spec_r. This fact is important to pass on so that Leto may verify the identical invariant in the outer loop.
- **upset(r, r2, spec_r, Ax, Ax2, spec_Ax).** This invariant enforces that in the event of an error during this outer loop iteration, the algorithm computes at least one

of set of instructions (`r` and `Ax` or `r2` and `Ax2`) correctly. Otherwise, the algorithm computes both sets correctly. It is an invariant in the middle loop because it must hold at the top of the inner loop.

- **`trans()`.** As before, this invariant passes the semantics of the fault model down into the inner loop so Leto knows that if an error already occurred in an earlier outer loop iteration, then another cannot occur in the inner loop.

Inner Loop. Verification then proceeds to the inner loop on Line 26. This loop contains the following invariants:

- **`upset(r, r2, spec_r, Ax, Ax2, spec_Ax)`.** This invariant enforces that in the event of an error during this outer loop iteration, the algorithm computes at least one of set of instructions (`r` and `Ax` or `r2` and `Ax2`) correctly. Otherwise, the algorithm computes both sets correctly. This invariant captures information about the impacts of errors on variables modified in the inner loop and passes this information back to the middle loop so that it may verify the `outer(r, spec_r)` invariant.

- **`trans()`.** As before, this invariant passes in the semantics of the fault model so Leto knows that if an error already occurred in an earlier outer loop iteration, then another cannot occur in this loop.

Assertion. Verification concludes with the assertion on Line 45. This assertion verifies that `r<r> == spec_r`, which enforces that `r<r>` is the correct residual and it satisfies Equation 9.5.

Chapter 10

Related Work

In addition to the work cited in the introduction on application-specific fault tolerance mechanisms and empirical techniques for reasoning about such mechanisms, Leto's contributions relate to the following work:

Reasoning about Approximate/Unreliable Computation. Researchers have developed a number of programming systems that enable developers to reason about *approximate computations*: computations for which the underlying execution substrate (e.g., the programming system and/or hardware system) augment the behavior of the application to produce approximate results. Typical goals of these systems are to trade accuracy of the overall computation for increased performance, reduced energy-consumption, or increased system availability. A major challenge with these systems is reasoning about the behavior of the resulting application. Researchers have developed a number of programming systems that enable different forms of reasoning. For example, EnerJ [52] and FlexJava [43] enable developers to demonstrate non-interference between approximate computations and critical parts of the computation that should not be modified. Rely [13], Chisel [36], and Decaf [7], enable developers to reason about the *reliability* of their applications: the probability that they produce the correct result.

The relaxed programming model [12] enables developers to prove both safety and accuracy programs for explicitly relaxed computations. The work I present in this theisis builds upon the relaxed programming model by using an asymmetric relational Hoare Logic to verify a

relaxed semantics of a program. However, I have additionally augmented Leto's language with first-class fault models that enable developers to specify complicated fault models that potentially span multiple operations in the program. In an additional contrast to the relaxed programming system, Leto also provides automation for many aspects of the proof where as the the relaxed programming system required manual, Coq-based proofs.

Meola and Walker propose a sub-structural logic for reasoning about fault tolerant programs [34]. The logic of their development enables the proof system to count the number of faults that have occurred and therefore reason about properties that may hold, for example, for a single-event upset model but not for a double-event upset model. Leto, provides a more expressive logic that supports reasoning about more complicated properties of the state of the fault model.

Relational Hoare Logic. Researchers have proposed a number of relational Hoare Logics and verification systems to support verifying relational properties of programs [4, 5, 12, 28, 60]. Typical properties of interest are 2-safety properties [61], such as equivalence checking (e.g., translation validation), determinism checking, and – in the case of relaxed programs – safety and accuracy properties. The verification algorithms produced by Sousa and Dillig [60] and Lahiri et al. [28] demonstrate that it's possible to automatically compose proofs for relational verification. Leto's verification system differs from that of CHL in that:

- The semantics of the two program executions are asymmetric.
- Leto attempts to verify with a specific program composition strategy that matches the types of proofs that are seen in practice for approximate and unreliably executed programs.

Namely, although the semantics of the two executions of the program differ, their structure is typically identical. Further, a major design point in Leto is that the verification of `assert` and `assume` statements in the relaxed execution can rely on assumed properties of the reliable execution. Leto therefore collects these properties from the reliable execution before verifying the relaxed execution.

Programming Models for Self-Stabilization. Researchers have proposed programming systems for verifying self-stability. For example, Self-Stabilizing Java provides developers with a type system and analysis that enables a developer to prove that any corruptible state of the program exists the system in a finite amount of time. Such algorithms are self-stabilizing in that they are guaranteed to return to valid state in bounded time. Leto's rich logic (versus the information-flow type system of Self-Stabilizing Java) enables developers to specify the richer invariants that need to be true of emerging algorithms for self-stability. For example, instead of verifying that corrupted state leaves the system within bounded time, Leto enables a developer verify that the corruption of the program's state is small enough that the algorithm's correction steps will work as designed.

Fault Rate Analysis I have driven the design and implementation of Leto by the anticipated fault rates, abstract fault models, and resilience tools exported by the computer architecture community. Namely, soft fault rates have led major organizations – such as Intel [27, 37–39], Google [69], NASA [23], DOE [59], and DARPA [1] – to express concern over such faults. Leto is the first system – to my knowledge – to enable automated verification for these faults.

The assumption of instruction-level arithmetic errors is the most common model for building:

- Application-specific fault analyses and mechanisms [10, 20, 21, 40, 50, 51, 54–56].
- Software-level fault tolerance analyses and mechanisms [30, 49, 53, 68, 70].
- Micro-architectural resilience analyses and mechanisms [3, 32, 33].
- Circuit-level resilience analyses/mechanisms [8, 9, 24, 31, 44, 45, 63].

Mitra et al. have found that combinational logic faults account for 11% of all soft errors [37]. In addition, soft error rates, including combinational faults, are expected to increase as chips continue grow in the number of transistors [37, 57]. These trends have inspired a variety of different contributions, including modeling the propagation of transient faults [15, 41], analyzing the rate of combinational soft faults [11, 47, 67, 71], analyzing the impact of combinational soft faults [46], and correcting combinational logic faults [38].

Chapter 11

Conclusion

Emerging computational platforms are increasingly vulnerable to errors. Future computations designed to execute on these platforms must therefore be designed to be *fault tolerant* and naturally resilient to error. I present a verification system, Leto, that facilitates the verification of application-specific fault tolerance mechanisms under programmer-specified execution models. As these proofs frequently relate a faulty execution to a fault-free one, Leto provides assertions that enable the developer to specify and verify expressions that relate the semantics of both executions. First-class execution models permit developers to convey information about the class of faults they expect their computational platforms to experience. By giving developers tools to verify relational invariants under first-class execution models, I enable developers to verify the self-stability of their programs.

Bibliography

[1] Saman Amarasinghe, Dan Campbell, William Carlson, Andrew Chien, William Dally, Elmootazbellah Elnohazy, Robert Harrison, William Harrod, Jon Hiller, Sherman Karp, Charles Koelbel, David Koester, Peter Kogge, John Levesque, Daniel Reed, Robert Schreiber, Mark Richards, Al Scarpelli, John Shalf, Allan Snavely, and Thomas Sterling. Exascale software study: Software challenges in extreme scale systems, 2009.

[2] JEDEC Solid State Technology Association et al. Jedec standard: Ddr4 sdram. *JESD79-4, Sep*, 2012.

[3] Todd M Austin. Diva: A reliable substrate for deep submicron microarchitecture design. MICRO, 1999.

[4] G. Barthe, J. Crespo, and C. Kunz. Relational verification using product programs. FM, 2011.

[5] N. Benton. Simple relational correctness proofs for static analyses and program transformations. POPL, 2004.

[6] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6), 2005.

[7] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. OOPSLA, 2015.

[8] Keith A Bowman, James W Tschanz, Nam Sung Kim, Janice C Lee, Chris B Wilkerson, Shih-Lien L Lu, Tanay Karnik, and Vivek K De. Energy-efficient and metastability-

immune resilient circuits for dynamic variation tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):49–63, 2009.

[9] Keith A Bowman, James W Tschanz, Shih-Lien L Lu, Paolo A Aseron, Muhammad M Khellah, Arijit Raychowdhury, Bibiche M Geuskens, Carlos Tokunaga, Chris B Wilkerson, Tanay Karnik, and Vivek K De. A 45 nm resilient microprocessor core for dynamic variation tolerance. *IEEE Journal of Solid-State Circuits*, 46(1):194–208, 2011.

[10] Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. ICS, 2008.

[11] S Buchner, M Baze, D Brown, D McMorrow, and J Melinger. Comparison of error rates in combinational and sequential logic. *IEEE transactions on Nuclear Science*, 44(6): 2209–2216, 1997.

[12] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. PLDI, 2012.

[13] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. OOPSLA, 2013.

[14] Michael Carbin and Martin C. Rinard. Automatically identifying critical input regions and code in applications. ISSTA, 2010.

[15] Liang Chen and Mehdi B Tahoori. An efficient probability framework for error propagation and correlation estimation. IOLTS, 2012.

[16] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. PPoPP, 2012.

[17] Cormac Flanagan and K Rustan M Leino. Houdini, an annotation assistant for esc/java. FME, 2001.

[18] Shyamnath Gollakota, Matthew Reynolds, Joshua Smith, and David Wetherall. The emergence of rf-powered computing. *Computer*, 47(1), January 2014.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL `http://doi.acm.org/10.1145/363235.363259`.

[20] Mark Hoemmen and Michael A Heroux. Fault-tolerant iterative methods via selective reliability. SC, 2011.

[21] Kuang-Hua Huang and Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[22] Ryan H.-M. Huang and Charles H.-P. Wen. Advanced soft-error-rate (ser) estimation with striking-time and multi-cycle effects. DAC, 2014.

[23] Allan H Johnston. Scaling and technology issues for soft error rates. 2000.

[24] Lee Hsiao-Heng Kelin, Lilja Klas, Bounasser Mounaim, Relangi Prasanthi, Ivan R Linscott, Umran S Inan, and Mitra Subhasish. Leap: Layout design through error-aware transistor positioning for soft-error resilient sequential cell design. IRPS, 2010.

[25] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in dram memories. *IEEE Computer Architecture Letters*, 14 (1):9–12, 2015.

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. ISCA, 2014.

[27] Nasser A Kurd, Subramani Bhamidipati, Christopher Mozak, Jeffrey L Miller, Timothy M Wilson, Mahadev Nemani, and Muntaquim Chowdhury. Westmere: A family of 32nm ia processors. ISSCC, 2010.

[28] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. CAV, 2012.

[29] M Lanteigne. How rowhammer could be used to exploit weaknesses in computer hardware, 2016.

[30] Tuo Li, Jude Angelo Ambrose, Roshan Ragel, and Sri Parameswaran. Processor design for soft errors: Challenges and state of the art. *ACM Computing Surveys (CSUR)*, 49 (3):57, 2016.

[31] K Lilja, M Bounasser, S-J Wen, R Wong, J Holst, N Gaspard, S Jagannathan, D Loveless, and B Bhuva. Single-event performance and layout optimization of flip-flops in a 28-nm bulk technology. *IEEE Transactions on Nuclear Science*, 60(4):2782–2788, 2013.

[32] David J. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, 31(7):681–685, 1982.

[33] Albert Meixner, Michael E Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. MICRO, 2007.

[34] Matthew L. Meola and David Walker. Faulty logic: Reasoning about fault tolerant programs. ESOP, 2010.

[35] G. Miremadi, J. Harlsson, U. Gunneflo, and J. Torin. Two software techniques for on-line error detection. FTCS, 1992.

[36] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. Chisel: reliability-and accuracy-aware optimization of approximate computational kernels. OOPSLA, 2014.

[37] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005.

[38] Subhasish Mitra, Ming Zhang, Saad Waqas, Norbert Seifert, Balkaran Gill, and Kee Sup Kim. Combinational logic soft error correction. ESOP, 2006.

[39] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. MICRO, 2003.

[40] Fabian Oboril, Mehdi B Tahoori, Vincent Heuveline, Dimitar Lukarski, and Jan-Philipp Weiss. Numerical defect correction as an algorithm-based fault tolerance technique for iterative solvers. PRDC, 2011.

[41] Martin Omana, Giacinto Papasso, Daniele Rossi, and Cecilia Metra. A model for transient fault propagation in combinatorial logic. IOLTS, 2003.

[42] Joseph A. Paradiso and Thad Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1), January 2005.

[43] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. Flexjava: Language support for safe and modular approximate programming. FSE, 2015.

[44] RC Quinn, JS Kauppila, TD Loveless, JA Maharrey, JD Rowe, ML Alles, BL Bhuva, RA Reed, M Mounasser, K Lilja, and LW Massengill. Frequency trends observed in 32nm soi flip-flops and combinational logic. *IEEE Transactions on Nuclear Science*, 2015.

[45] RC Quinn, JS Kauppila, TD Loveless, JA Maharrey, JD Rowe, MW McCurdy, EX Zhang, ML Alles, BL Bhuva, RA Reed, WT Holman, M Bounasser, K Lilja, and LW Massengill. Heavy ion seu test data for 32nm soi flip-flops. REDW, 2015.

[46] R Rajaraman, JS Kim, Narayanan Vijaykrishnan, Yuan Xie, and Mary Jane Irwin. Seat-la: A soft error analysis tool for combinational logic. VLSI Design, 2006.

[47] Rajeev R Rao, Kaviraj Chopra, David T Blaauw, and Dennis M Sylvester. Computing the soft error rate of a combinational logic circuit using parameterized descriptors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26 (3):468–479, 2007.

[48] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. NIPS, 2011.

[49] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. Swift: Software implemented fault tolerance. CGO, 2005.

[50] Amber Roy-Chowdhury and Prithviraj Banerjee. Algorithm-based fault location and recovery for matrix computations. FTCS, 1994.

[51] Amber Roy-Chowdhury and Prithviraj Banerjee. Algorithm-based fault location and recovery for matrix computations on multiprocessor systems. *IEEE transactions on computers*, 45(11):1239–1247, 1996.

[52] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. PLDI, 2011.

[53] Thiago Santini, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. Effectiveness of software-based hardening for radiation-induced soft errors in real-time operating systems. ARCS, 2017.

[54] Piyush Sao and Richard Vuduc. Self-stabilizing iterative solvers. ScalA, 2013.

[55] Piyush Sao, Oded Green, Chirag Jain, and Richard Vuduc. A self-correcting connected components algorithm. FTXS, 2016.

[56] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. ICS, 2012.

[57] Premkishore Shivakumar, Michael Kistler, Stephen W Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. DSN, 2002.

[58] Martin L Shooman. N-modular redundancy. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*, pages 145–201, 2002.

[59] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[60] M. Sousa and I. Dillig. Cartesian hoare logic for verifying k-safety properties. PLDI, 2016.

[61] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. SA5, 2005.

[62] Anna Thomas and Karthik Pattabiraman. Error detector placement for soft computing applications. *ACM Trans. Embed. Comput. Syst.*, 15(1):8:1–8:25, January 2016. ISSN 1539-9087. doi: 10.1145/2801154. URL `http://doi.acm.org/10.1145/2801154`.

[63] M Turowski, K Lilja, K Rodbell, and P Oldiges. 32nm soi sram and latch seu crosssections measured (heavy ion data) and determined with simulations. SEE, 2015.

[64] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. MICRO, 2016.

[65] Sriram Krishnamoorthy Vishal Chandra Sharma, Ganesh Gopalakrishnan. Towards reseiliency evaluation of vector programs. DPDNS, 2016.

[66] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.

[67] Feng Wang and Yuan Xie. Soft error rate analysis for combinational logic using an accurate electrical masking model. *IEEE Transactions on Dependable and Secure Computing*, 8(1):137–146, 2011.

[68] Jiesheng Wei and Karthik Pattabiraman. Blockwatch: Leveraging similarity in parallel programs for error detection. DSN, 2012.

[69] Keun Soo Yim. Characterization of impact of transient faults and detection of data corruption errors in large-scale n-body programs using graphics processing units. IPDPS, 2014.

[70] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Hauberk: Lightweight silent data corruption error detector for gpgpu. IPDPS, 2011.

[71] Ming Zhang and Naresh R Shanbhag. Soft-error-rate-analysis (sera) methodology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10): 2140–2155, 2006.

Appendix A

Full Semantics

In this appendix I present the full dynamic semantics of the Leto language. I have elided the rules presented in Figure 3-2 as they remain unchanged. A preprocessing pass performs the following actions:

- It places all variables without a label into a reliable memory region.
- It flattens multidimensional vectors into single dimensional vectors.
- It inlines function calls.

The alloc function I use in DECLARE and DECLARE-MATRIX takes a mapping from variables to addresses σ and an integer n and returns the first addresses in a contiguous block of n unmapped addresses in σ.

Figures A-1 and A-2 expand on the operational semantics present in Figure 3-2.

READ-MATRIX

$$\frac{a = \sigma(x) \qquad n_o = h(o(r_2)) \qquad n = h(a + n_o) \qquad q = \theta(a) \qquad \langle m, \texttt{read}, (n, q)\rangle \Downarrow_\mu \langle n',\ m'\rangle}{\langle r_1 = x[r_2], \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \xrightarrow{\mu} \langle r_1 = n', \langle \sigma,\ h,\ \theta,\ m'\rangle\rangle}$$

WRITE-MATRIX

$$\frac{\begin{array}{c} a = \sigma(x) \qquad n_o = h(o(r_2)) \\ n_{old} = h(a + n_o) \qquad n_{new} = \sigma(r_1) \qquad q = \theta(a) \qquad \langle m, \texttt{write}, (n_{old}, n_{new}, q)\rangle \Downarrow_\mu \langle n_r,\ m'\rangle \end{array}}{\langle x[r_2] = r_1, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \xrightarrow{\mu} \langle \texttt{skip}, \langle \sigma,\ h[(a + n_o) \mapsto n_r],\ \theta,\ m'\rangle\rangle}$$

RELATIONAL-ASSERT

$$\frac{}{\langle \texttt{assert_r}\ r, \varepsilon\rangle \xrightarrow{\mu} \langle \texttt{skip}, \varepsilon\rangle}$$

DECLARE

$$\frac{a = \mathrm{alloc}(\sigma, 1)}{\langle \texttt{@region}(q)\ \tau\ x, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \xrightarrow{\mu} \langle \texttt{skip}, \langle \sigma[x \mapsto a],\ h[a \mapsto 0],\ \theta[a \mapsto q],\ m\rangle\rangle}$$

DECLARE-MATRIX

$$\frac{a = \mathrm{alloc}(\sigma, n) \qquad h' = h[a \mapsto 0] \ldots [(a + n - 1) \mapsto 0]}{\langle \texttt{@region}(q)\ \texttt{matrix<}\tau\texttt{>}\ x(n), \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \xrightarrow{\mu} \langle \texttt{skip}, \langle \sigma[x \mapsto a],\ h',\ \theta[a \mapsto q],\ m\rangle\rangle}$$

FORALL-T

$$\frac{\begin{array}{c} a = \mathrm{alloc}(\sigma, 1) \\ \forall n. \langle b, \langle \sigma[x \mapsto a],\ h[a \mapsto n],\ \theta[a \mapsto \texttt{reliable}],\ m\rangle\rangle \Downarrow_\mu \langle \mathit{true}, \langle \sigma[x \mapsto a],\ h[a \mapsto n],\ \theta[a \mapsto \texttt{reliable}],\ m\rangle\rangle \end{array}}{\langle \texttt{forall}(\tau\ x)(b), \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \Downarrow_\mu \langle \mathit{true}, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle}$$

FORALL-F

$$\frac{\begin{array}{c} a = \mathrm{alloc}(\sigma, 1) \\ \exists n. \langle b, \langle \sigma[x \mapsto a],\ h[a \mapsto n],\ \theta[a \mapsto \texttt{reliable}],\ m\rangle\rangle \Downarrow_\mu \langle \mathit{false}, \langle \sigma[x \mapsto a],\ h[a \mapsto n],\ \theta[a \mapsto \texttt{reliable}],\ m\rangle\rangle \end{array}}{\langle \texttt{forall}(\tau\ x)(b), \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \Downarrow_\mu \langle \mathit{false}, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle}$$

EXISTS-T

$$\frac{\begin{array}{c} a = \mathrm{alloc}(\sigma, 1) \\ \exists n. \langle b, \langle \sigma[x \mapsto a],\ h[a \mapsto n],\ \theta[a \mapsto \texttt{reliable}],\ m\rangle\rangle \Downarrow_\mu \langle \mathit{true}, \langle \sigma[x \mapsto a],\ h[a \mapsto n],\ \theta[a \mapsto \texttt{reliable}],\ m\rangle\rangle \end{array}}{\langle \texttt{exists}(\tau\ x)(b), \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \Downarrow_\mu \langle \mathit{true}, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle}$$

EXISTS-F

$$\frac{\begin{array}{c} a = \mathrm{alloc}(\sigma, 1) \\ \forall n. \langle b, \langle \sigma[x \mapsto a],\ h[a \mapsto n],\ \theta[a \mapsto \texttt{reliable}],\ m\rangle\rangle \Downarrow_\mu \langle \mathit{false}, \langle \sigma[x \mapsto a],\ h[a \mapsto n],\ \theta[a \mapsto \texttt{reliable}],\ m\rangle\rangle \end{array}}{\langle \texttt{exists}(\tau\ x)(b), \langle \sigma,\ h,\ \theta,\ m\rangle\rangle \Downarrow_\mu \langle \mathit{false}, \langle \sigma,\ h,\ \theta,\ m\rangle\rangle}$$

Figure A-1: Extension to Dynamic Language Semantics

M-DECLARE

$$\frac{}{\langle \tau\ x, \langle h,\ o\rangle\rangle \rightarrow \langle \mathtt{skip}, \langle h[x \mapsto 0],\ o\rangle\rangle}$$

M-ASSIGN

$$\frac{\langle e, \langle h,\ o\rangle\rangle \Downarrow \langle n, \langle h,\ o\rangle\rangle}{\langle x = e, \langle h,\ o\rangle\rangle \rightarrow \langle \mathtt{skip}, \langle h[x \mapsto n],\ o\rangle\rangle}$$

M-DECLARE-OP

$$\frac{}{\langle \mathtt{operator} \oplus x_1^* \ \mathtt{when}\ p_w \ \mathtt{modifies}\ x_2^* \ \mathtt{ensures}\ p_e, \langle h,\ o\rangle\rangle \rightarrow \langle \mathtt{skip}, \langle h,\ o :: \langle \oplus,\ x_1^*,\ p_w,\ p_e\rangle\rangle\rangle}$$

F-READ

$$\frac{\begin{array}{c} o = \mu(\mathtt{read}, [x], P_w, P_e) \\ o \in q \qquad m[x_1 \mapsto n_1] \models P_w \qquad m'[x_1 \mapsto n_1][\mathit{result} \mapsto n_2] \models P_e \qquad \mathrm{dom}(m) = \mathrm{dom}(m') \end{array}}{\langle \mathtt{model.read}(n_1, q),\ m\rangle \Downarrow_\mu \langle n_2,\ m'\rangle}$$

F-WRITE

$$\frac{\begin{array}{c} o = \mu(\mathtt{write}, [x_1, x_2], P_w, P_e) \qquad o \in q \\ m[x_1 \mapsto n_{old}][x_2 \mapsto n_{new}] \models P_w \qquad m'[x_1 \mapsto n_3][x_2 \mapsto n_{new}] \models P_e \qquad \mathrm{dom}(m) = \mathrm{dom}(m') \end{array}}{\langle \mathtt{model.write}(n_{old}, n_{new}, q),\ m\rangle \Downarrow_\mu \langle n_3,\ m'\rangle}$$

Figure A-2: Extension to Dynamic Execution Model Semantics

Appendix B

Full Self-stabilizing Conjugate Gradient Descent Implementation

I present my full implementation for self-stabilizing conjugate gradient descent below.

```
1  const real SQR_MIN_MAX_AIJ = 2;
2  
3  property_r sqr_lt(matrix<real> v, int i) :
4    ((q<r>[i<r>] - q<o>[i<o>]) * (q<r>[i<r>] - q<o>[i<o>])) <
5        SQR_MIN_MAX_AIJ;
6  
7  property_r dmr_eq(matrix<real> x1,
8                    matrix<real> x2,
9                    matrix<real> spec_x) :
10   x1<r> == spec_x && x2<r> == spec_x;
11 
12 property_r dmr_imp(matrix<real> x1,
13                    matrix<real> x2,
14                    matrix<real> spec_x) :
15   (x1<r> == x2<r>) -> (x1<r> == spec_x);
16 
17 requires 0 < N
18 r_requires eq(N) && eq(M) && eq(F) && eq(A)
19 matrix<real> ss_cg(int N,
20                    int M,
```

```
21                    int F,
22                    matrix<real> A(N, N),
23                    matrix<real> b(N),
24                    matrix<real> x(N)) {
25
26  matrix<real> r(N), p(N), q(N), next_x(N), next_r(N), next_p(N)
27  real alpha, beta, tmp, tmp2, num, denom;
28  int man_mod;
29
30  matrix<real> zeros(N);
31  @noinf for (int i = 0; i < N; ++i) { zeros[i] = 0 };
32
33  int it = 0;
34
35  @noinf for (int i = 0; i < N; ++i) {
36    tmp = 0;
37    @noinf for (int j = 0 ; j < N; ++j) {
38      tmp = tmp + A[i][j] * x[i];
39    }
40    r[i] = b[i] - tmp;
41  }
42  p = r;
43
44  @noinf @label(outer_while)
45  while (it < M)
46        invariant 0 < N
47        invariant_r eq(A) && eq(it) && eq(M) &&
48                    eq(N) && eq(man_mod) && eq(F) {
49    if (man_mod == F) {
50      matrix<real> r2(N), q2(N);
51      specvar matrix<real> spec_r(N), spec_q(N);
52
53      r = r2 = spec_r = q = q2 = spec_q = zeros;
54      bool not_run = true;
55      @noinf while (not_run || r != r2 || q != q2)
56                 invariant_r !model.upset -> (dmr_eq(r, r2, spec_r)&&
```

```
57                                                         dmr_eq(q, q2, spec_q))
58                   invariant_r dmr_imp(r, r2, spec_r)
59                   invariant_r dmr_imp(q, q2, spec_q) {
60         not_run = false;
61 
62         for (int i = 0; i < N; ++i) {
63           for (int j = 0; j < N; ++j) (1 == 1) (1 == 1) {
64             tmp = A[i][j] *. x[j];
65             tmp2 = A[i][j] *. x[j];
66             specvar real spec_tmp = A[i][j] * x[j];
67             r[i] = r[i] +. tmp;
68             r2[i] = r2[i] +. tmp2;
69             spec_r[i] = spec_r[i] + spec_tmp;
70 
71             tmp = A[i][j] *. p[j];
72             tmp2 = A[i][j] *. p[j];
73             spec_tmp = A[i][j] * p[j];
74             q[i] = q[i] +. tmp;
75             q2[i] = q2[i] +. tmp2;
76             spec_q[i] = spec_q[i] + spec_tmp;
77           }
78         }
79       }
80 
81       assert_r(!outer_while[model.upset] -> r<r> == spec_r);
82       assert_r(!outer_while[model.upset] -> q<r> == spec_q);
83 
84       @noinf for (int i = 0; i < N; ++i) { r[i] = b[i] - r[i]; }
85 
86       num = 0;
87       denom = 0;
88       @noinf for (int i = 0; i < N; ++i) {
89         tmp = r[i] * p[i];
90         num = num + tmp;
91         denom = p[i] * q[i];
92       }
```

```
93        alpha = num / denom;
94
95        @noinf for (i = 0; i  < N; ++i) {
96          tmp = alpha * p[i];
97          next_x[i] = x[i] + tmp;
98          tmp = alpha * q[i];
99          next_r[i] = r[i] - tmp;
100       }
101
102       num = 0;
103       denom = 0;
104       @noinf for (int i = 0; i < N; ++i) {
105         tmp = -next_r[i];
106         tmp = tmp * q[i];
107         num = num + tmp;
108
109         tmp = p[i] * q[i];
110         denom = denom + tmp;
111       }
112       beta = num / denom;
113
114       @noinf for (i = 0; i < N; ++i) {
115         tmp = beta * p[i];
116         next_p[i] = next_r[i] + tmp;
117       }
118     } else {
119       for (int i = 0; i < N; ++i)
120           invariant 0 <= i <= N {
121         q[i] = 0;
122         @label(inner_err)
123         for (int j = 0; j < N; ++j)
124             invariant 0 <= j <= N && 0 <= i < N
125             invariant_r (model.upset == false && eq(p)) ->
126                 q<r>[i<r>] == q<o>[i<o>] {
127           tmp = A[i][j] *. p[j];
128           q[i] = q[i] +. tmp;
```

```
129
130            assert_r((!inner_err[model.upset] && eq(p)) -> sqr_lt(q, i));
131          }
132        }
133
134        num = 0;
135        denom = 0;
136        @noinf for (int i = 0; i < N; ++i) {
137          tmp = r[i] * r[i];
138          num = num + tmp;
139          denom = p[i] * q[i];
140        }
141        alpha = num / denom;
142
143        @noinf for (i = 0; i  < N; ++i) {
144          tmp = alpha * p[i];
145          next_x[i] = x[i] + tmp;
146          tmp = alpha * q[i];
147          next_r[i] = r[i] - tmp;
148        }
149
150        num = 0;
151        denom = 0;
152        @noinf for (int i = 0; i < N; ++i) {
153          num = next_r[i] * next_r[i];
154          denom = r[i] * r[i];
155        }
156        beta = num / denom;
157
158        @noinf for (i = 0; i < N; ++i) {
159          tmp = beta * p[i];
160          next_p[i] = next_r[i] + tmp;
161        }
162      }
163      ++it;
164
```

```
165        p = next_p;
166        x = next_x;
167        r = next_r;
168
169
170        ++man_mod;
171
172        if (man_mod == M) {
173          man_mod = 0;
174        }
175      }
176      return x;
177    }
```