Leto: Verifying Application-Specific Hardware Fault Tolerance with Programmable Execution Models

BRETT BOSTON, Massachusetts Institute of Technology, USA
ZOE GONG, Massachusetts Institute of Technology, USA
MICHAEL CARBIN, Massachusetts Institute of Technology, USA

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 we present Leto, an SMT-based automatic verification system that enables developers to verify their applications with respect to an 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. In this paper, we present the Leto programming language and its corresponding verification system. We also demonstrate Leto on several applications that leverage application-specific fault tolerance mechanisms.

CCS Concepts: • **Computer systems organization → Reliability**; • **Software and its engineering → Formal software verification**;

Additional Key Words and Phrases: Approximate Computing

ACM Reference Format:
Brett Boston, Zoe Gong, and Michael Carbin. 2018. Leto: Verifying Application-Specific Hardware Fault Tolerance with Programmable Execution Models. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 163 (November 2018), 30 pages. https://doi.org/10.1145/3276533

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) [Amarasinghe et al. 2009; Borkar 2005; Johnston 2000; Kurd et al. 2010; Mitra et al. 2005, 2006; Mukherjee et al. 2003; Shivakumar et al. 2002; Yim 2014].

Authors' addresses: Brett Boston, Massachusetts Institute of Technology, Cambridge, MA, USA, boston@csail.mit.edu; Zoe Gong, Massachusetts Institute of Technology, Cambridge, MA, USA, zoegong@mit.edu; Michael Carbin, Massachusetts Institute of Technology, Cambridge, MA, USA, mcarbin@csail.mit.edu.

2475-1421/2018/11-ART163
https://doi.org/10.1145/3276533

Challenges. These systems encounter *faults*—anomalies in the underlying physical device—that produce *errors*—unanticipated or incorrect values that are visible to the program. Simple fault models include bit flips in the output of arithmetic, logical, and memory operations. These faults can be *transient*—occurring nondeterministically with the device eventually returning to correct behavior—or *permanent*—with the device never returning to correct behavior. A key challenge for building applications for these platforms is that reasoning about the reliability of these applications requires reasoning about the semantics of the platform and its impact on the application's behavior.

1.1 Application-Specific Fault Tolerance

In response to this increased error vulnerability, researchers have expanded on historical results for algorithm-based fault tolerance [Bronevetsky and de Supinski 2008; Hoemmen and Heroux 2011; Huang and Abraham 1984; Oboril et al. 2011; Roy-Chowdhury and Banerjee 1994, 1996; Sao et al. 2016; Sao and Vuduc 2013; Shantharam et al. 2012], 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. These mechanisms take the original application and produce a *protected* version of the application that includes runtime error detection and recovery mechanisms.

For example, application-specific fault tolerance techniques for linear algebra instrument the application with lightweight checksums that validate if the computation produced the correct results. For some applications, these checksums are exact. However, for other applications, these checksums either are not known to exist or, at best, compromise on their error coverage.

Other techniques include selective *n*-modular redundancy in which a developer either manually or with the support of a dynamic 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 [Carbin and Rinard 2010; Thomas and Pattabiraman 2016; Venkatagiri et al. 2016; Vishal Chandra Sharma 2016]. Another class of techniques is fault-tolerant algorithms that through the addition of algorithm-specific checking and correction code are tolerant to faults [Du et al. 2012; Hoemmen and Heroux 2011; Sao et al. 2016; Sao and Vuduc 2013].

Challenges. A major barrier to implementing these techniques is that their results either rely on empirical guarantees or—for self-stabilizing algorithms—hinge on the assumption that the underlying computing substrate's fault model matches the assumptions of the algorithmic formalization.

1.2 Verified Application-Specific Fault Tolerance

To address these challenges we present Leto, an SMT based, automatic verification system that supports reasoning about unreliably executed programs. Leto enables a developer to verify their application-specific fault tolerance mechanism by providing tools to 1) programmatically specify the behavior of the computing substrate's fault model and 2) verify *relational* assertions that relate the behavior of the unreliably executed program to that of the reliable execution. Specifically, Leto automatically weaves the behavior of the underlying hardware system—as given by a specification—into the execution of the main program. In addition, Leto's program logic 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.

1.2.1 Execution Models. Leto permits developers to programmatically specify an execution model. Figure 1 presents a Leto specification for a *single-event upset* (SEU) execution model with specifications for multiplication. An SEU model is a common *fault model* that application developers in the area of fault tolerance use to model the execution behavior of an application such that they can provide a variety of fault tolerance mechanisms [Chen et al. 2008; Yim et al. 2010]. The underlying assumption is that faults in a system (e.g., those due to cosmic radiation) occur with a probability such that at most one fault will occur during the execution of an application.

```
1 bool upset = false;
2
3 operator *(real x1, real x2)
4    ensures (result == x1 * x2);
5
6 operator *(real x1, real x2)
7    when (!upset)
8    modifies (upset)
9    ensures (upset);
```

Fig. 1. Unbounded SEU Execution Model

The model exports two versions of the multiplication operator. Line 3 specifies the standard reliable implementation of multiplication. The model denotes this fact with its `ensures` clause which asserts what must be true of the model's state and outputs after execution of the operation. This operation specifically constrains the value of `result`—which represents the result of the operation—to equal `x1 * x2` where `*` has standard multiplication semantics. Line 6 specifies an unreliable implementation of the multiplication operator. This implementation does not place any constraints on `result` and therefore permits arbitrary, unbounded errors in the operation's result.

Stateful. Because the model is an SEU model it must track whether a fault has occurred so that the model exposes at most one fault to the application. This model is therefore *stateful* and to achieve this semantics, the model includes a boolean valued state variable—`upset` (Line 1)—that records whether or not a fault has already occurred during the execution of the program. The model additionally predicates the availability of its unreliable operations by a *guard*. Specifically, an operation's guard is the optionally-specified boolean expression that occurs after the `when` keyword. The `when` clause for the unreliable version requires `!upset` indicating that the unreliable version is only enabled if a fault has yet to occur. Leto models the dynamic execution of the application such that the execution exposes only enabled operation implementations at a given program point.

1.2.2 Acceptability Properties. Leto enables developers to automatically verify the basic goal of an application-specific fault tolerance mechanism: ensure that the resulting application satisfies its *acceptability properties* [Carbin et al. 2012], such as its *safety* and *accuracy*.

Safety Properties: 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 of 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 typically seen in verification systems.

Accuracy Properties: 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 `abs(x<o> - x<r>) < epsilon` in Leto specifies that the difference in value of `x` between the program's original, reliable execution (denoted by `x<o>`) and relaxed execution (denoted by `x<r>`) is at most `epsilon`.

Execution-Specific Properties. Given an execution model, Leto also enables developers to refer to the execution model's state. For example, in many self-stabilizing iterative algorithms, the proof of convergence for the algorithm in the presence of faults requires reasoning about three cases: 1) the portion of the execution in which no fault has occurred, 2) the iteration on which a fault occurs (assuming an SEU model), and 3) the portion of the execution after the fault. Leto enables developers to verify such properties by exposing the state of the fault model into the program logic.

Asymmetric Relational Verification. Leto provides and implements an Asymmetric Relational Hoare Logic [Carbin et al. 2012] as its core program logic. An Asymmetric Relational Hoare Logic is a variant of the standard Hoare Logic that natively refers to the values of variables between two executions of the program. Leto's use of a relational program logic serves two goals: 1) it gives a semantics to accuracy properties and 2) 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 its safety properties, then relaxed executions of the application with its deployed application-specific fault tolerance mechanisms also satisfy these safety properties.

1.3 Contributions

This paper presents the following contributions:

Execution Models. We present a language for specifying execution models that provides stateful, input-dependent selection of each operation's implementation. We demonstrate numerous sample execution models.

Programming Language and Semantics. We present language constructs that enable developers to specify assertions that refer to the state of the execution model. These constructs enable a developer to, for example, specify the precise properties that self-stabilizing applications require to verify high-level convergence properties.

Program Logic and Verification Algorithm. Leto's program logic enables developers to lower the overhead of verifying a standard safety property by enabling techniques that demonstrate the non-interference between the application's faults and the validity of a property. Leto's verification algorithm additionally automates this process through the inclusion of loop invariant inference.

Case Studies. We evaluate Leto on several self-correcting algorithms (Jacobi, Self-stabilizing Conjugate Gradient, Self-stabilizing Steepest Descent, and Self-correcting Connected Components) and demonstrate that it is possible to verify the key invariants required to prove that these algorithms' self-stability guarantees hold for their implementations. We consider execution models that capture a range of substrates, including emerging hardware systems that bound potential error, emerging hardware security vulnerabilities (Rowhammer [Kim et al. 2014]), as well as standard fault modeling assumptions that expose unbounded errors to the application.

Leto's contributions enable developers to specify and verify the rich properties seen in applications with application-specific fault tolerance mechanisms. Within the developer's workflow, Leto therefore aids developers in building concrete implementations of their applications by enabling them to first specify and verify their fault tolerance mechanisms within Leto's environment.

2 EXAMPLE: VERIFYING HARDWARE FAULT TOLERANCE

Verifying hardware fault tolerance is challenging task in which the end goal is to deliver a fault tolerance scheme that balances fault coverage (the set of faults for which the computation produces an acceptable result) with the added overhead of instrumentation for potentially detecting and recovering from the fault. In this section, we illustrate this task's challenges as well as how to address them with Leto. We first present two fault tolerance approaches for a simplified vector-vector product that we then combine to produce a self-stabilizing version of the Jacobi iterative method [Jacobi 1845] using Leto.

```
vector<real> product(uint N, vector<real> x(N), vector<real> y(N))
{
   vector<real> result(N);

   for (uint i = 0; i < N; ++i) { result[i] = x[i] * y[i]; }

   return result;
}
```

Fig. 2. Vector Product

```
property_r eq_array(vector<real> x, uint N) :
   ∀(uint i)((i < N<r>) -> (x<o>[i] == x<r>[i]));

requires_r eq(N) && eq(x) && eq(y)
vector<real> product(uint N, vector<real> x(N), vector<real> y(N))
{
   vector<real> result(N);

   for (uint i = 0; i < N; ++i)
      invariant_r eq_array(result, i)
   {
      real d_1 = x[i] *. y[i];

      real d_2 = x[i] *. y[i];
      if (d_1 != d_2) {
         d_1 = x[i] *. y[i];
      }

      result[i] = d_1;
   }

   assert_r(eq_array(result, N));

   return result;
}
```

Fig. 3. Vector Product Under SEU

Vector-Vector Product. Figure 2 presents an implementation of a vector-vector (Hadamard/-Pointwise) product in Leto. We start with this simple example because vector products are components in many numerical algorithms and exist in three of our four benchmarks as an intermediate calculation in matrix-vector products.

This implementation defines a function `product` that takes a size parameter `N`, a vector of real numbers, `x`, of size `N`, a vector `y` of size `N`, and returns the pointwise product of `x` and `y`. This computation uses Leto's `real` datatype to communicate that the values should be modeled as real numbers in the theories of the system's underlying SMT solver (Z3).[1]

2.1 Exact Fault Tolerance through Dual-Modular Redundancy

Figure 3 presents a vector-vector product implementation protected using dual-modular redundancy (DMR) such that the resulting implementation is exact in the presence of errors generated by the unbounded single-event upset model specified in Figure 1 of Section 1. Specifically, Leto verifies that this implementation when executed under the model satisfies its stated invariants.

Two features in 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 program's standard, *original execution* and its alternative *relaxed execution* under a specified execution model.

[1]Leto also supports traditional `float` data types. However, `real` can serve as a simplifying assumption and optimization over the complexity of Z3's floating-point implementation.

Relaxed Execution. Leto exports relaxed operations by enabling developers to specify that an operation may execute according to the execution model specification (versus a standard implementation) by appending a dot to the operation such as the multiplication operation '`*.`' on Line 12, Line 14, and Line 16. The first multiplication corresponds to the multiplication on Line 5 in the original implementation, with the additional two operations corresponding to the implementation's hardware fault tolerance scheme.

Fault Tolerance. A hardware fault tolerance scheme typically has two components: the code that corresponds to detecting that an error has occurred and the code the recovers from the detected error. Dual modular redundancy (DMR) prescribes a simple implementations of these components.

Detection. In DMR, to detect an error the developer or system executes a redundant copy of the computation. In this case, the developer inserts a redundant multiply on (Line 14). Note that this multiply is also relaxed, with the typical assumption in DMR being that redundant computation executes under the same fault assumptions.

The implementation then checks if the duplicated value matches the original value. If not, DMR prescribes that the computation should then execute its recovery mechanism.

Recovery. The simplest recovery mechanism for DMR is to simply re-execute the operation. In this case, the implementation re-executes the multiplication on Line 16 (again with the same fault assumptions), setting `d_1` to the correct value.

Reasoning. The DMR detection and recovery scheme is correct because of the single-event upset model. Specifically, at most one multiplication during the execution of `product` can return an incorrect result. Therefore we know that if either multiplication on Line 12 or Line 14 return a correct result, then the equality comparison must identify that a fault occurred. Further, given that correct multiplications are deterministic, the recovery multiplication on Line 16 only executes if a fault occurred previously. Second, given that the only admissible fault has already occurred, the recovery multiplication must execute correctly, therefore restoring `d_1` to the correct value.

Relational Verification. To verify that this vector product implementation produces the correct result in Leto, the developer uses the `assert_r` statement on Line 22 to assert `eq_array(result, N)`. `eq_array` is a *relational assertion*. This implementation defines `eq_array` as a *property* – a hygienic macro in Leto that enables code reuse within loop invariants and assertions – on Line 1. This property takes a vector `x` and a size `N` and enforces that for every index `i`, `x<o>[i] == x<r>[i]`, where `x<o>` refers to the value of `x` in the standard, original execution of the program and `x<r>` refers to the value of `x` in the relaxed execution. To facilitate the verification of this condition we also include it in the loop invariant on Line 9. Another necessary component to verifying this assertion is that `N`, `x`, and `y` have the same values in both the original and relaxed executions. The implementation enforces this property through the relational function precondition (`requires_r`) on Line 4. Leto expands terms of the form `eq(x)` to `x<o> == x<r>`.

Verification Algorithm. 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. 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. For example, in this program, Leto infers `eq(N)`, `eq(x)`, `eq(y)` and `i <= N`, which is necessary to demonstrate that all of the vector accesses are in bounds.

Summary. DMR is a traditional and simple scheme for hardware fault tolerance. However, even in its simplicity, there are still challenges. For example, consider if the developer were to execute their application on the Leto execution model in Figure 4, which produces two faults instead one. (Leto supports the old keyword in the spirit of similar constructs in ESC/Java [Flanagan and Leino 2001], Spec# [Barnett et al. 2004], and other verification systems, which denotes the value of the corresponding variable before the operator executes.) For this model, DMR is no longer correct: both d_1 and d_2 could result in the same incorrect result and therefore the error detection check would erroneously pass, leading to an undetected error.

```
uint upset = 2;

operator *(real x1, real x2)
   ensures (result == x1 * x2);

operator *(real x1, real x2)
   when (upset > 0)
   modifies (upset)
   ensures (upset = old(upset) - 1);
```

Fig. 4. Double-Event Upset Execution Model

Alternatively, the developer may be willing to risk two incorrect executions under the assumption that errors are independent (Figure 5) and, therefore, the probability that two instructions produce the same incorrect result is small enough to not be a concern . However, in this case, the recovery multiplication must then execute reliably (via a separate fully reliable hardware substrate [Sampson et al. 2011] or additional replication) because it is then possible for the implementation to correctly detect the first error, but then have the second error corrupt the multiplication used for recovery.

```
uint upset = 2;
real first_upset_value = 0;

operator *(real x1, real x2)
   ensures (result == x1 * x2);

operator *(real x1, real x2)
   when (upset == 2)
   modifies (upset, first_upset_value)
   ensures (upset = 1 &&
            first_upset_value == result);

operator *(real x1, real x2)
   when (upset == 1)
   modifies (upset)
   ensures (upset = 0 &&
            result != first_upset_value);
```

Fig. 5. Independent Double-Event Upset Execution Model

These considerations and scenarios demonstrate that even simple, traditional hardware fault tolerance schemes with seemingly straightforward implementations have subtle interactions between their behavior and the execution model.

2.2 Approximate Computation through Bounded Error

Another hardware fault tolerance approach is to permit a computation to produce approximate results [Carbin et al. 2013b; Sampson et al. 2011]. In this case, the developer does not necessarily seek to produce the exact result in the event of an error; instead the developer understands that the end user of a component's or application's results can tolerate error. For example, many self-healing iterative algorithms for solving linear systems of equations can tolerate uncorrected errors in their intermediate computations and still converge to correct solutions. The typical outcome of such an error, however, is that the solver takes longer (than an error free execution) to compute an answer. However, if the developer can determine and control the frequency and magnitude of these errors, then they can also control the increase in convergence time [Sao and Vuduc 2013].

For example, the Jacobi iterative method has the property that the change in the number of iterations to converge after an error is bounded logarithmically by the magnitude of the error. Thus, by bounding the magnitude of errors, a developer using the Jacobi method can derive a static bound on the maximum impact errors can have on convergence time.

```
1  const real E_REL = ...;
2  bool upset = false;
3
4  operator *(real x1, real x2)
5     ensures (result == x1 * x2);
6
7  operator *(real x1, real x2)
8     when !upset && (0 != x1 * x2)
9     modifies (upset)
10    ensures upset && -E_REL <= 1 - result / (x1 * x2) <= E_REL;
```

Fig. 6. SEU Execution Model with Relative Error

```
1  const real max = ...;
2  const real eps = ...;
3
4  property_r bounded_diff(vector<real> x, uint N) :
5     ∀(uint i)((i < N<r>) -> (abs(x<o>[i] - x<r>[i]) < eps));
6
7  requires_r eq(N) && eq(x) && eq(y)
8  requires forall(uint fi)(abs(x[fi]) < max && abs(y[fi]) < max)
9  matrix<real> product(uint N, matrix<real> x(N), matrix<real> y(N))
10 {
11   matrix<real> result(N);
12
13   for (uint i = 0; i < N; ++i)
14         invariant_r bounded_diff(result, i)
15   {
16     result[i] = x[i] *. y[i];
17   }
18
19   assert_r(bounded_diff(result, N));
20
21   return result;
22 }
```

Fig. 7. Vector Product Under SEU with Relative Error

Bounded Error Multiplication. Figure 6 presents an single-event upset model that also provides a relative error bound on the result of an errant multiply. This model corresponds to the guarantees provided by hardware designs that, for example, are protected by truncated error correction [Sullivan and Swartzlander 2012, 2013].

The model exports two versions of the multiplication operator. Line 4 specifies the standard reliable implementation of multiplication. Line 7 additionally specifies an unreliable implementation for the case where `x1 * x2` is non-zero. The semantics of this unreliable operator guarantees that even in the presence of an error, the result is within `E_REL` percent of the original result. To implement the bound, the developer places a bound on the results of the multiplication using inequalities over the `result` variable. In this case, `E_REL` (defined on Line 1) is the constant that specifies the magnitude of the error bound.

Bounded Error Vector Product. Figure 7 presents a vector product implementation annotated to verify under the relative error execution model. All of the relaxation in this implementation occurs in the loop (Line 14) where the execution model may corrupt the value of `result[i]`. On Line 19 the implementation uses a relational assertion (`assert_r`) to assert that `eps` (a constant) is an upper bound on the impact of these errors (Line 5).

Verifying this vector product implementation relies on Leto's specification capabilities to establish bounds on the error in the product. To verify that this vector product implementation has bounded error the developer uses the `assert_r` statement on Line 19 to assert `bounded_diff(result, N)`.

We define the `bounded_diff` property on Line 4. This property takes a vector `x` and a size `N` and enforces that for every index `i`, the absolute error defined by `x<o>[i] - x<r>[i]` is less than `eps`. To facilitate the verification of this condition we also include it in the loop invariant on Line 14.

In addition to the preconditions on `N`, `x`, and `y` as before (Figure 3, Line 4), this version requires each component of the two input vectors to be less than `max` (Line 8). Because `bounded_diff` asserts a bound on the absolute deviation in each component of `result` whereas the execution model provides a relative bound, this additional precondition enables the solver to determine that applying that relative bound to each array value results in a value that satisfies the absolute bound (for satisfying values of `max` and `model.E_REL`).

Summary. Using Leto, a developer can bound the error that occurs in this approximate vector product. For example, because at most one fault happens during `product`'s execution, at most one position in `result` differs from the value in a reliable execution and, therefore, the norm of the difference between `result` in the relaxed program versus the original program is at most `eps`. Such bounded error properties are important for a variety of approximate computations [Carbin et al. 2012; Chaudhuri et al. 2011]. Moreover, for self-stabilizing iterative solvers, this property enables the developer to bound the additional time to convergence given faults in the underlying hardware.

2.3 Application-Specific Fault Tolerance

Figure 8 presents an implementation of the Jacobi iterative method, alternatively Jacobi, in Leto. The Jacobi iterative method is an algorithm for solving a system of linear equations. Specifically, given a matrix of coefficients `A` and a vector `b` of intercepts, the algorithm computes a solution vector, `x`, where $\mathtt{A} * \mathtt{x} = \mathtt{b}$. 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 from 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*. Specifically, if faults do not modify the contents of `A`, then Jacobi is in a *valid state* at the end of each iteration: if no additional faults occur during its 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, Jacobi enjoys the nice result that the change in the number of iterations required to converge from the new starting point is bounded logarithmically by the magnitude of the error in the solution vector.

Verifying Jacobi for a given execution platform therefore poses two challenges: 1) verifying that faults only affect the value of `x` and 2) 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.

```
 1  const real E = ...;
 2
 3  property_r sig(real sum) :
 4    (!model.upset -> eq(sum)) &&
 5    (!out[model.upset] && model.upset -> (abs(sum<o> - sum<r>) < E));
 6
 7  uint N; int iters;
 8  matrix<real> A(N,N); vector<real> b(N); vector<real> x(N);
 9
10  @label(out)
11  for ( ; 0 <= iters; --iters) invariant_r !model.upset -> eq(x)
12  {
13     vector<real> next_x(N);
14
15     for (uint i = 0; i < N; ++i)
16       invariant_r !model.upset -> eq(next_x)
17       invariant_r !out[model.upset] & model.upset -> bounded_diff(next_x,N)
18     {
19        real sum = 0;
20
21        for (uint j = 0; j < N; ++j) invariant_r sig(sum)
22        {
23           if (i != j) {
24              real delta = A[i][j] *. x[j];
25
26              if (E/model.E_REL-E <= abs(delta)) {
27                 delta = A[i][j] * x[j];
28              }
29
30             sum = sum + delta;
31           }
32
33        }
34        real num = b[i] - sum;
35        next_x[i] = num / A[i][i];
36     }
37
38     x = next_x;
39
40     assert_r eq(A);
41     assert_r(bounded_diff(x, N));
42  }
```

Fig. 8. Jacobi Iterative Method

Jacobi Implementation. The overall architecture of the implementation in Figure 8 is that the outer loop on Line 11 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 15 iterates through each x_i (stored at `x[i]`), sums the other terms in the ith equation using the third loop (Line 21) into `sum`, and then computes `x[i]` as the value `b[i] - sum/A[i][i]`. The property `bounded_diff` is the same as before (Figure 7). We discuss `sig` below.

Relaxation. All of the relaxation in Jacobi occurs in the third loop (Line 21), where the relaxed execution may perturb the value of `sum`. Specifically, the implementation performs a relaxed multiplication (Line 24) for each calculation of `delta`.

To ensure that the overall calculation of `sum` does not exceed the error bound `E`, the implementation dynamically asserts that each `delta` calculation is sufficiently accurate. To achieve this, the implementation dynamically checks that `delta` is less than `E / model.E_REL - E`, which asserts that `delta` is small enough that even if the calculation were to have encountered the worst-case possible error according to the specification for the relaxed multiplication, then the resulting error in `sum` still would not exceed `E`. If this check succeeds, then the implementation continues to the next iteration of the loop. If this check fails, then the implementation falls back to a reliable multiplication implementation (Line 27), enabling the computation to recover from the error.

Self-Stability. To verify that this Jacobi implementation is self-stabilizing the developer uses the `assert_r` statement on Line 40 to assert `eq(A)`, which denotes that `A` has the same value in both the original and relaxed execution. This property therefore asserts that faults do not disturb the matrix of coefficients and therefore precludes any execution models that may disturb `A`.

Convergence Bound. Jacobi also enjoys a bound on the additional number of iterations added to its execution given a fault. Specifically, $\Delta_c = O(\log_T(\frac{1}{(N \times \mathtt{E})^2}))$ where Δ_c is the number of additional iterations in the relaxed execution, N is the size of the `x` vector, `E` is the maximum perturbation in each element of the solution vector to due a fault in an iteration, and T is a value between 0 and 1 representing the magnitude of the non-diagonal elements of A relative to the magnitude of the diagonal elements of A. We specify that the maximum perturbation is bounded by `E` with the `assert_r` on Line 41, which asserts that `bounded_diff(x, N)`.

Verification Approach. To verify Jacobi, the developer needs to provide a set of loop invariants that structure the proof. In Jacobi, there two key invariants.

The developer specifies one invariant for the outermost loop on Line 11: `!model.upset -> eq(x)`. Given our target execution model in Figure 6, this invariant therefore states that if a fault has 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 compute the same result. By default, Leto models the two executions as starting from the same state. Therefore, all variables initialized in Line 7 and Line 8 have the same values between the two executions. An important observation here is that developer-driven reasoning is guided by the model's state.

The invariant on Line 17 is a key step towards the main proof goal in that it bounds the difference in `next_x` if a fault occurs on the current iteration of the loop. In Leto, the notation `!out[model.upset]` is a reference to the state of the execution model at the last execution of the *label* `out`, which corresponds to the beginning of the body of the outer `while` loop (Line 11). Leto's support for labels enables the developer and the verification system to refer to the value of a model state at some control flow point in the program. This capability is important because, as shown in the previous invariant and here, the model state communicates whether any faults have occurred and, moreover, enables the developer (and system) to reason about whether a fault has occurred between two program points. In this invariant, labels enable the developer to assert that the bound applies for this single iteration of the iterative algorithm where the fault occurs.

Finally, the invariant `sig(sum)` bounds the total error on `sum`. This property is true because if a fault did not occur on the previous calculation of `x` in the outer loop then at most one fault may happen during the calculation of each `delta` that contributes to `sum` and the error in each `delta` is bounded by `E`. Leto verifies that the unreliable multiplication in combination with the conservative check on Line 26 establishes this fact.

Summary. In Jacobi, these properties work together to enable the developer to refine the dual modular redundancy approach (as explored in Section 2.1) to eliminate the redundant multiplication required to detect and error while still providing an useful bound on the behavior of the application. Leto's relational verification features therefore enable a developer to navigate the space between exact fault tolerance, approximate computation, and appplication-specific fault tolerance, which combines detection and correction logic with application-specific properties to produce more efficient detection and correction schemes.

$$c \in \text{constants},\ x \in \text{variables},\ r \in \text{memory regions}$$

$$\begin{aligned}
S &\rightarrow (\texttt{@region}\ r)^?\ \tau\ x \mid (\texttt{@region}\ r)^?\ \tau\ x\ (c^+) \mid x = E \\
&\mid S\ ;\ S \mid \texttt{assume}\ P \mid \texttt{assert\_r}\ P_r \mid \texttt{assert}\ P \\
&\mid \texttt{if}\ (B)\ \{\ S\ \}\ \texttt{else}\ \{\ S\ \} \mid \texttt{while}\ (B)\ I^*\ \{\ S\ \}
\end{aligned}$$

$$\begin{aligned}
\tau &\rightarrow \texttt{int} \mid \texttt{real} \mid \texttt{bool} \\
I &\rightarrow \texttt{invariant}\ P \mid \texttt{invariant\_r}\ P_r
\end{aligned}$$

$$M \rightarrow \texttt{model}\ \{\ x^*\ O^+\ \} \qquad O \rightarrow \texttt{operator}\ Op\ (\tau\ x)^+\ C^+$$

$$\begin{aligned}
Op &\rightarrow \neg \mid \neg. \mid \oplus \mid \prec \mid \diamond \mid \texttt{read} \mid \texttt{write} \\
C &\rightarrow \texttt{when}\ P \mid \texttt{ensures}\ P \mid \texttt{modifies}\ x^+
\end{aligned}$$

$$\begin{aligned}
\diamond &\rightarrow \wedge \mid \wedge. \mid \rightarrow \mid \rightarrow. \mid \ldots \\
\prec &\rightarrow < \mid <. \mid = \mid =. \mid \ldots \\
\oplus &\rightarrow + \mid +. \mid \times \mid \times. \mid \ldots
\end{aligned}$$

$$\begin{aligned}
E &\rightarrow x \mid E \oplus E \mid \texttt{model}.x \mid x[E] \\
E_r &\rightarrow x\texttt{<o>} \mid x\texttt{<r>} \mid E_r \oplus E_r \\
&\mid x\texttt{<o>}[E_r] \mid x\texttt{<r>}[E_r] \mid E
\end{aligned}$$

$$\begin{aligned}
B &\rightarrow \mathit{true} \mid \mathit{false} \mid E \prec E \\
&\mid B \diamond B \mid f\ E^* \mid \neg B \mid \neg. B \\
P &\rightarrow \forall x\ B \mid \exists x\ B \mid B \\
P_r &\rightarrow \mathit{true} \mid \mathit{false} \mid E_r \prec E_r \mid P_r \diamond P_r \\
&\mid \neg P_r \mid \neg. P_r \mid \forall x\ P_r \mid \exists x\ P_r
\end{aligned}$$

Fig. 9. Language Syntax

3 LANGUAGE

Figure 9 presents the core of Leto's programming language. Leto provides a general-purpose imperative language that includes specification primitives (e.g., `requires`) in the spirit of ESC/-Java [Flanagan and Leino 2001], Boogie [Barnett et al. 2005], Eiffel [Meyer 1992], and Spec# [Barnett et al. 2004] to support verifying applications.

Programs. A program consists of a sequence of statements S. Although we also support functions, we elide them here.

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 and writes may have a custom semantics according to the execution model.

Memory Regions. In addition to binary operators, Leto permits the specification of read and write behavior. Figure 10 presents an adaptation of the single-event upset model in Figure 1 to memory. Read and write specifications may contain an additional `@region` annotation that enables developers to partition their program variables into multiple memory regions with differing read and write characteristics. The annotations on Line 3 and Line 7 denote that each respective specification gives a semantics to variables stored in the `unreliable` memory region. In general, the set of regions is not fixed in Leto; a developer can define arbitrary regions by name.

```
1  bool upset = false;
2
3  @region(unreliable)
4  write(uint dest, uint src)
5     ensures (dest == src);
6
7  @region(unreliable)
8  write(uint dest, uint src)
9     when (!upset)
10    modifies (upset)
11    ensures (upset);
```

Fig. 10. Single-Event Upset Memory Model

When Leto encounters an expression of the form `v = e` where `v` is in the memory region `unreliable`, Leto substitutes occurrences of `dest` in the model with `v` and occurrences of `src` with `e`. Line 3 specifies a reliable write operator while Line 7 specifies a faulty write operator that writes arbitrary values to memory, with the result being that the system stores an erroneous value in the variable represented by `dest` and subsequent reads of that variable return the erroneous value.

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.

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 operate with the semantics as given in the program's execution model.

Assertions and Assumptions. Leto also enables developers 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, P, 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, P_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`, and `for` statements. For `while` and `for` statements, a developer can 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 1) the target operator for the specification, 2) a list of variables as parameters to the specification, and 3) 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 fault 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

$$\begin{array}{lcl} x & \in & \text{Var} \\ r & \in & \text{Reg} \\ n & \in & \text{Int}_\text{N} \end{array} \qquad \begin{array}{rcl} \mathit{Exp} & \rightarrow & n \mid r \oplus r \\ S & \rightarrow & r = \mathit{Exp} \mid r = x \mid x = r \\ & \mid & \texttt{assert}\ r \mid \texttt{assume}\ r \\ & \mid & \texttt{skip} \mid S\ ;\ S \mid \texttt{if}\ r\ S\ S \\ & \mid & \texttt{while}\ r\ P_r^*\ S \end{array}$$

Fig. 11. Syntax of Lowered Language (Abbreviated)

We next present an abbreviated dynamic semantics of Leto's language. We expand on these semantics in our companion technical report [Boston et al. 2018]. We formalize the semantics via a lowered syntax of the Leto language that includes registers (Figure 11). We assume a standard compilation process that translates high-level Leto to this lowered language. Additionally, to support our formalization of Leto's program logic, we extend the predicates P and P_r with registers into P^* and P_r^*, respectively.

3.1.1 Preliminaries. 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: 1) a frame maps a program variable to the address of the memory region allocated for that variable in the heap, and 2) a frame maps a register to its current value. The model state stores the values for state variables within the execution model.

Frames, Heaps, Model States, Environments. A *frame*, $\sigma \in \Sigma = \text{Var} \cup \text{Reg} \rightarrow \text{Int}_\text{N}$, is a finite map from variables and registers to N-bit integers. A *heap*, $h \in H = \text{Loc} \rightarrow \text{Int}_\text{N}$, 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*, $m \in M = \text{Var} \longrightarrow \text{Int}_\text{N}$, is a finite map from model state variables to N-bit integer values. An *environment*, $\varepsilon \in \text{E} = \Sigma \times H \times \Theta \times M$, is a tuple consisting of a frame, a heap, a region map, and a model state. An *execution model specification*, $\mu \subseteq Op \times list(\text{Var}) \times set(\text{Var}) \times P \times P$, is a relation consisting of tuples of an operation $op \in Op$, a list of input variables, a set of modified variables, and two unary logical predicates representing the when and ensures clauses of the operation.

Initialization. For clarity of presentation, we 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. Here we provide an abbreviated presentation of the execution model relation $\langle m, op, (args) \rangle \Downarrow_\mu \langle n,\ m' \rangle$. 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.

$$\frac{\begin{array}{c}\mu(\oplus, [x_1, x_2], X, P_w, P_e) \qquad \forall x_m \in X \cdot \text{fresh}(x'_m) \qquad m[x_1 \mapsto n_1][x_2 \mapsto n_2] \models P_w \\ m'[x_1 \mapsto n_1][x_2 \mapsto n_2][\forall x_m \in X \cdot x_m \mapsto x'_m][result \mapsto n_3] \models P_e \qquad \text{dom}(m) = \text{dom}(m')\end{array}}{\langle m, \oplus, (n_1, n_2) \rangle \Downarrow_\mu \langle n_3,\ m' \rangle}\ \textsc{f-binop}$$

The [F-BINOP] rule specifies the meaning of this relation for binary operations. This relation states that the value of an operation $\oplus$ given a tuple of input values (n_1, n_2) and an execution model state m evaluates to value n_3 and a new model state m'. The rule relies on the relation $\mu(op, vlist, X, P_w, P_e)$ which specifies the list of argument names, *vlist*, the set of modified variables X, the *precondition* P_w, and the *postcondition* P_e for the operation *op* in the developer-provided execution model. The set of modified variables is the union of the `modifies` clauses in the operation's specification. 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 nondeterministically selects an operation specification, result value, and output model state subject to the constraint that: 1) the current model state satisfies the precondition (after the inputs to the operation have been appropriately assigned into the model state), 2) the output model state satisfies the postcondition (after the inputs, modified variables, and result value have been appropriately assigned into the model state), and 3) 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 (e.g., reads and writes) is similar with the sole distinction being the number of arguments passed to the operation. For clarity of presentation, we elide the presentation of rules for those operations.

3.1.3 Language Semantics. We next present the nondeterministic 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 statements is largely similar to that of traditional approaches except for the statements' ability to encounter faults. We categorize Leto's instructions into four categories: *register instructions*, *assertions*, *memory instructions*, 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.

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

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

[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.

Assertions. The `assert` and `assume` 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.

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

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

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 nondeterministically 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.

Control Flow. The rules for control flow 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 [Sampson et al. 2011; Vishal Chandra Sharma 2016].

Big-Step Semantics. To support the formalization in the remainder of the paper, we introduce the big-step relation $\langle s, \varepsilon \rangle \Downarrow_m v \subseteq S \times \mathrm{E} \times V$ where $V ::= \mathrm{E} \mid \texttt{fail}\ \mathrm{E}$ such that $\langle s, \varepsilon \rangle \Downarrow v$ is the reflexive transitive closure of $\rightarrow_m$ that yields the environment ε if execution ends successfully in a `skip` statement or yields the pair `fail` ε when the execution ends in a failure. We also introduce the big-step relation $\langle s, \varepsilon \rangle \Downarrow v \subseteq S \times \mathrm{E} \times V$ where $\langle s, \varepsilon \rangle \Downarrow v \equiv \langle s, \varepsilon \rangle \Downarrow_\rho v$ where ρ denotes a *fully reliable* fault model where the only implementations exposed for each operation are fully reliable implementations.

4 PROGRAM LOGIC

Leto's program logic is a relational program logic in that it relates relaxed executions of the program to its original, reliable execution. A key idea behind our development is the separation of the rules into a part that solely characterizes the reliable execution of the program, (the Left Rules), a part that solely characterizes the relaxed execution (the Right Rules), and a part the characterizes the lockstep execution of the reliable and relaxed execution (the Lockstep Rules). The result is an Asymmetric Relational Hoare Logic that characterizes the two interpretations of the program.

4.1 Preliminaries

Assertion Logic Syntax and Semantics. Figure 9 presents our language syntax, including the syntax of our assertion language. Assertions include standard quantified boolean predicates, P^*, with the standard semantic function $[\![P^*]\!] \in \mathcal{P}(\mathrm{E})$ that gives the denotation of P^* as the set of environments that satisfy the predicate. Assertions also include quantified *relational* boolean predicates, P_r^*, with the semantic function $[\![P_r^*]\!] \in \mathcal{P}(\mathrm{E} \times \mathrm{E})$ that gives P_r^* the meaning of the set of pairs of environments that satisfy the predicate. In our standard convention, the first environment of the pair corresponds to the state of the reliable execution whereas the second environment corresponds to that of the relaxed execution.

Auxiliary Definitions. To support the formalization in the remainder of the paper we define the auxiliary notation $inj_t(\cdot)$ 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.

4.2 Proof Rules

Figures 12 and 13 provide an abbreviated presentation of the rules of our program logic. We present the remainder of the rules in our companion technical report [Boston et al. 2018]. We have partitioned the presentation into two parts: 1) the Left Rules and Right Rules for primitive statements and 2) the Lockstep Rules.

Left Rules. The Left Rules, which we denote by the judgment $\vdash_l \{\ P_r^*\ \}\ s\ \{\ Q_r^*\ \}$, characterize the behavior of the reliable execution of the statement s. The denotation of the judgment is that if $(\varepsilon_1, \varepsilon_2) \models P_r^*$, and $\langle s, \varepsilon_1 \rangle \Downarrow \varepsilon_1'$, then $(\varepsilon_1', \varepsilon_2) \models Q_r^*$. 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.

Right Rules. The right rules, which we denote by the judgment $\mu \vdash_r \{\ P_r^*\ \}\ s\ \{\ Q_r^*\ \}$, characterize the behavior of the relaxed execution of s under a fault model specification μ. The denotation of the judgment is similar to that of the Left Rules: if $(\varepsilon_1, \varepsilon_2) \models P_r^*$, $\langle s, \varepsilon_2 \rangle \Downarrow_\mu \varepsilon_2'$, then $(\varepsilon_1, \varepsilon_2') \models Q_r^*$.

$$\textsc{assign-l}\quad \frac{}{\vdash_l \{\ Q_r^*[n/inj_o(r)]\ \}\ r = n\ \{\ Q_r^*\ \}} \qquad \textsc{assign-r}\quad \frac{}{\mu \vdash_r \{\ Q_r^*[n/inj_r(r)]\ \}\ r = n\ \{\ Q_r^*\ \}}$$

$$\textsc{binop-l}\quad \frac{}{\vdash_l \{\ Q_r^*[inj_o(r_1 \oplus r_2)/inj_o(r)]\ \}\ r = r_1 \oplus r_2\ \{\ Q_r^*\ \}}$$

$$\textsc{binop-r}\quad \frac{\begin{array}{c}\text{fresh}(r') \qquad \exists([x_1, x_2], X, P_w^*, P_e^*) \in \mu(\oplus) \cdot inj_r(P_w^*[r_1/x_1][r_2/x_2]) \\ Q_r'^* = \left(\bigvee_{([x_1,x_2],X,P_w^*,P_e^*)\in\mu(\oplus)} inj_r(P_w^*[r_1/x_1][r_2/x_2]) \longrightarrow inj_r(P_e^*[r_1/x_1][r_2/x2][\forall x_m \in X \cdot \text{fresh}(x_m')/x_m][r'/result])\right)\end{array}}{\mu \vdash_r \{\ Q_r^*[inj_r(r')/inj_r(r)] \wedge Q_r'^*\ \}\ r = r_1 \oplus r_2\ \{\ Q_r^*\ \}}$$

$$\textsc{assert-l}\quad \frac{}{\vdash_l \{\ true\ \}\ \texttt{assert}\ r\ \{\ inj_o(r)\ \}} \qquad \textsc{assert-r}\quad \frac{}{\mu \vdash_r \{\ inj_r(r)\ \}\ \texttt{assert}\ r\ \{\ inj_r(r)\ \}}$$

$$\textsc{assume-l}\quad \frac{}{\vdash_l \{\ true\ \}\ \texttt{assume}\ r\ \{\ inj_o(r)\ \}} \qquad \textsc{assume-r}\quad \frac{\mu \vdash_r \{\ P_r^*\ \}\ \texttt{assert}\ r\ \{\ Q_r^*\ \}}{\mu \vdash_r \{\ P_r^*\ \}\ \texttt{assume}\ r\ \{\ Q_r^*\ \}}$$

Fig. 12. Left and Right Rules for Primitive Statements

$$\boxed{\vdash \{\ P_r^*\ \}\ s\ \{\ Q_r^*\ \}} \qquad \textsc{split}\quad \frac{\mu \vdash \{\ P_r^*\ \}\ s \sim s\ \{\ Q_r^*\ \}}{\mu \vdash \{\ P_r^*\ \}\ s\ \{\ Q_r^*\ \}} \qquad \textsc{seq}\quad \frac{\mu \vdash \{\ P_r^*\ \}\ s_1\ \{\ R_r^*\ \} \qquad \mu \vdash \{\ R_r^*\ \}\ s_2\ \{\ Q_r^*\ \}}{\mu \vdash \{\ P_r^*\ \}\ s_1\ ;\ s_2\ \{\ Q_r^*\ \}}$$

$$\textsc{if}\quad \frac{\begin{array}{c} b \equiv r = true \qquad \mu \vdash \{\ P_r^* \wedge inj_o(b) \wedge inj_r(b)\ \}\ s_1\ \{\ Q_r^*\ \} \qquad \mu \vdash \{\ P_r^* \wedge \neg inj_o(b) \wedge inj_r(b)\ \}\ s_2 \sim_r s_1\ \{\ Q_r^*\ \} \\ \mu \vdash \{\ P_r^* \wedge inj_o(b) \wedge \neg inj_r(b)\ \}\ s_1 \sim_r s_2\ \{\ Q_r^*\ \} \qquad \mu \vdash \{\ P_r^* \wedge \neg inj_o(b) \wedge \neg inj_r(b)\ \}\ s_2\ \{\ Q_r^*\ \}\end{array}}{\vdash \{\ P_r^*\ \}\ \texttt{if}\ r\ s_1\ s_2\ \{\ Q_r^*\ \}}$$

Fig. 13. Lockstep Control Flow and Structural Rules

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 fault model specification μ terminates, then the resulting environment pair satisfies the proof's postcondition.

Lockstep Rules. The Lockstep Rules together constitute the main top-level judgment of the logic reasons about relations between the two semantics as they proceed in lockstep.: $\mu \vdash \{\ P_r^*\ \}\ s\ \{\ Q_r^*\ \}$. The denotation is that if $(\varepsilon_1, \varepsilon_2) \models P_r^*$, $\langle s, \varepsilon_1\rangle \Downarrow \varepsilon_1'$, and $\langle s, \varepsilon_2\rangle \Downarrow_\mu \varepsilon_2'$, then $(\varepsilon_1', \varepsilon_2') \models Q_r^*$.

4.2.1 *Left and Right Rules.*

Register Assignment. The rules [ASSIGN-L] and [ASSIGN-R] capture the semantics of the register assignment statement, $r = n$ in the lowered language. 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 [Hoare 1969]. 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. We note that given these results, assignment is reliable in both the reliable and relaxed executions with the primary distinction being which environment is modified (either that corresponding to the reliable execution or that of 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.

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. Specifically, while the logic requires that 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 an assertion assumes the validity of the assertion whereas the Right rule asserts. Although the `assert` and `assume` have the same semantics in the reliable and relaxed executions, the intentions of the statements differ. Specifically, if a developer places an `assert` in the program, the assumption is that they have used other means to evaluate the validity of that assertion in the reliable execution (potentially including other verification systems). An `assume` statement, however, does not carry that intention.

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 we must verify that faults do not interfere with the reasoning behind an assumption.

Control Flow. For clarity of presentation we have elided the left and right rules control flow because 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.

4.2.2 Lockstep Rules. To support the lock step rules, we first present the [STAGE] rule, which joins the Left Rules and Right Rules.

$$\textsc{stage}\quad \frac{\vdash_l \{\, P_r^* \,\}\ s_1\ \{\, R_r^* \,\} \qquad \mu \vdash_r \{\, R_r^* \,\}\ s_2\ \{\, Q_r^* \,\}}{\mu \vdash \{\, P_r^* \,\}\ s_1 \sim s_2\ \{\, Q_r^* \,\}} \qquad \textsc{inverse-stage}\quad \frac{\mu \vdash_r \{\, P_r^* \,\}\ s_2\ \{\, R_r^* \,\} \qquad \vdash_l \{\, R_r^* \,\}\ s_1\ \{\, Q_r^* \,\}}{\mu \vdash \{\, P_r^* \,\}\ s_1 \sim_r s_2\ \{\, Q_r^* \,\}}$$

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 we 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 R_r^*, before then applying the Right Rule for s_2 to R_r^*. The rule [SPLIT] provides a rationale for this specific composition. The rule [INVERSE-STAGE] has the opposite semantics, applying the Right Rule for s_2, yielding a new predicate R_r^*, before then applying the Left Rule for s_1 to R_r^*. The rule [IF] provides a rationale for this composition during non-lockstep execution.

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 to 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.

If. The rule [IF] gives the semantics of `if` statements. The rule considers all cases of the execution of the statement. Specifically, the reliable and relaxed executions may proceed in lockstep or they may diverge by proceeding down different branches. The logic captures this divergence by leveraging the inverse staging rule to apply the Right Rules for the branch on which the relaxed execution has taken before applying the Left Rules for the one which the reliable 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 relaxed execution takes before proceeding with the reliable execution.

4.3 Properties

Leto's program logic ensures two basic properties of Leto programs: *preservation* and *progress*. The preservation property states the partial correctness of the logic (but does not not establish termination—and therefore total correctness). The progress property establishes that the relaxed execution of a program verified with Leto 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. We state these properties formally below and provide proofs of these theorems in our companion technical report [Boston et al. 2018].

THEOREM 4.1 (PRESERVATION).
If $\mu \vdash \{\ P_r^*\ \}\ s\ \{\ Q_r^*\ \}$ *and* $(\varepsilon_1, \varepsilon_2) \models P_r^*$ *and* $\langle s, \varepsilon_1 \rangle \Downarrow \varepsilon_1'$ *and* $\langle s, \varepsilon_2 \rangle \Downarrow_\mu \varepsilon_2'$, *then* $(\varepsilon_1', \varepsilon_2') \models Q_r^*$

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.

THEOREM 4.2 (PROGRESS).
If $\mu \vdash \{\ P_r^*\ \}\ s\ \{\ Q_r^*\ \}$ *and* $(\varepsilon_1, \varepsilon_2) \models P_r^*$ *and* $\langle s, \varepsilon_1 \rangle \Downarrow \varepsilon_1'$ *and* $\langle s, \varepsilon_2 \rangle \Downarrow_\mu v_2$, *then* $\neg \mathit{failed}(v_2)$ *where* $\mathit{failed}(\langle \texttt{fail}, \varepsilon \rangle) = \mathit{true}$

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.

5 SYSTEM AND IMPLEMENTATION

The Leto system includes other features that increase user productivity, including invariant inference and model refinement.

Invariant Inference. One of Leto's key enabling features is loop invariant inference. Our approach uses Houdini-style [Flanagan and Leino 2001] template-based inference. Specifically, Leto seeds its inference process with inequalities that relate values in the original execution of the program to values in the relaxed execution. For example, as in other relational verification systems [Lahiri et al. 2012], if the program variable `x` is live on entry to a loop, then Leto seeds its inference algorithm with the conjunct `eq(x)`, denoting the equivalence of the value of `x` in both the original and relaxed executions. Where Leto differs from other techniques is through the adaptation of aging [Furia and Meyer 2010] to *model-driven versioning* of invariants. Specifically, invariants in our domain often need to reason about if the model state changes between two program points.

```
@label(out)
while (...)
invariant_r !model.upset -> eq(x)
{
  for (...) {
    ...
  }
}
```

Fig. 14. Jacobi Loop Structure

For example, Figure 14 presents an abbreviated snippet of the outer loop structure of Jacobi. The invariant `!model.upset -> eq(x)` is critical for supporting this proof in that if no error has occurred on entry into the loop, then `x` has the correct value inside of the body of the loop and can therefore be used to discharge proof obligations inside the inner loop. However, the property as stated may not necessarily hold in the inner loop because a fault may occur between the beginning of the outer loop and the point of use within the inner loop. Therefore, instead, what is needed is a *versioned* copy of this invariant: `!out[model.upset] -> eq(x)`, which asserts that if `model.upset` was `false` at the previous execution of the program point labeled by `out`, then `eq(x)` must hold.

Leto's invariant inference algorithm automatically applies model-driven versioning to propagate invariants that refer to the model's state from outer loops to inner loops, thereby lowering the overall verification burden. In our companion technical report, we present a detailed description of Leto's verification and loop invariant inference algorithms [Boston et al. 2018].

Refinement. To enable a parallel development process in which developers may successively build their hardware, models, and programs in tandem, Leto supports execution model *refinement*. Refinement enables developers to construct a lattice of models such that more precise submodels satisfy the specification of less precise supermodels.

```
1  refines seu;
2  import seu.reliable;
3
4  const real E_REL = ...;
5
6  @refines(unreliable)
7  operator *(real x1, real x2)
8     when !upset && (0 != x1 * x2)
9     modifies (upset)
10    ensures upset &&
11      -E_REL <= 1 - result / (x1 * x2) <= E_REL;
```

Fig. 15. Refined SEU Model

Figure 15 presents the relative error model refined (Figure 6) from the unbounded single-event upset model (Figure 1). Leto supports multiple refinement, allowing submodels to refine any number of supermodels. Line 2 imports the reliable operator from the seu model by name. To enable developers to import and refine operators by name, Leto supports optional labels on operators. Therefore, the developer would augment (Figure 1), such that each operator had the label `@label{reliable}` and `@label{unreliable}`, respectively. This makes these operators available to the submodel.

Developers may add additional operators by indicating that they refine a named operator in each supermodel. Leto checks that this submodel operator refines each supermodel operator by verifying that the `when` clause of the submodel operator logically implies the `when` clause of each supermodel operator and the `ensures` clause of the submodel operator logically implies the `ensures` clause of each supermodel operator. Every additional operator in a submodel must explicitly refine some named operator in each supermodel. When using multiple refinement, any imported operators from one model must also explicitly refine an operator in every other supermodel.

When refining a model, the submodel implicitly imports all variable declarations and initializations from the supermodel. The submodel may declare and initialize variables in its own namespace, but it may not modify the variable state of the supermodel with one exception: submodels may initialize uninitialized variables in the supermodel. We enforce this constraint on model variables because Leto programs may inspect, modify, and assert over model variables so there must be no statically discernible difference between the submodel and the supermodel state to a program written with knowledge of supermodel state variables.

By verifying refinement, Leto guarantees that programs verified under an abstract execution model will also verify under specialized versions of that model. Therefore, refinement separates model elaboration from program verification. That is, it provides an interface between hardware vendors and software developers. Software developers may verify their programs under loosely defined execution models, while hardware vendors may provide detailed models that are true to the underlying semantics of their hardware. As long as software developers use Leto to verify that these precise models refine their loose models, Leto guarantees that their programs will run as expected on the hardware vendor's product. Our results in Figure 18 in Section 7 illustrate instances where the same program and invariants verifies for different models.

Implementation. Leto's verification 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 from Section 4. For additional detail, we provide a full specification of Leto's verification algorithm in our companion technical report [Boston et al. 2018].

Leto generates constraints to be solved by Microsoft's Z3 SMT solver [De Moura and Bjørner 2008]. Our system makes use of Z3's real, int, and bool types as well as uninterpreted functions for arrays/matrices. Together, Leto's support for quantifiers, non-linear integer arithmetic, and its representation of matrices and vectors as uninterpreted functions can result in Z3 using undecidable theories for which it is not complete. The practical impact of this design is that it is possible for Z3 to be unable to verify valid constraints. However, we have been able to successfully verify critical fault tolerance properties for several applications as presented in the following section.

6 CASE STUDY: CONNECTED COMPONENTS WITH ROWHAMMER MODEL

Figure 16 presents an abbreviated presentation of a single iteration of the Self-Correcting Connected Components algorithm (SC-CC) [Sao et al. 2016], 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. One 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,

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

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 SC-CC detects an error at $CC^i[v]$, it repeats the computation for node v with reliably backed storage.

To elaborate, the semantics of self-stabilizing connected components is such that if a sufficiently large error happens, then the algorithm can easily detect and correct it due to its natural semantics.

```
assume (N < model.min_err);

@region(relaxed) uint next(N) = CC;

model.reliable = false;
for (uint i = 0; i < N; ++i) {
   for (uint j = 0; j < N; ++j) {
      if (next[i] <= i && adj[i][j]) {
         next[i] = min(next[i], CC[j]);
      }
   }
 }

model.reliable = true;
for (uint i = 0; i < N; ++i) {
   if (i < next[i]) {
      for (uint j = 0; j < N; ++j) {
         if (adj[i][j]) {
            next[i] = min(next[i], CC[j]);
         }
      }
   }
}
CC = next;
assert_r(eq(CC));
```

Fig. 16. Connected Components (Abbreviated)

Specifically, the label at each node is nonincreasing, therefore if a label increases between iterations, then the algorithm must have experienced an error.

The algorithm is therefore naturally resilient to certain types of errors (large errors) in the sense that there exists an efficient, application-specific error detectors for those errors. However, if errors can cause a label to decrease incorrectly, then the algorithm cannot easily detect the error. Therefore, if the hardware model delivers large errors, then the developer can use Leto to verify that the application always detects and corrects errors.

Rowhammer Model. Figure 17 presents a switchable Rowhammer execution model. This execution model simulates a Rowhammer attack that allows an attacker to selectively flip bits in DRAM by issuing frequent reads on DRAM rows [Kim et al. 2014]. Unlike most of the models we have presented thus far, this one enables developers to model memory errors. Additionally, it enables switchability, permitting the program to selectively disable errors to emulate selective Rowhammer protection techniques [Aweke et al. 2016]. Line 4 specifies a reliable write operator while Line 8 specifies a faulty write operator. The faulty write operator introduces errors that are larger than `err_min`.

```
1  const uint err_min = ...;
2  bool reliable = false;
3
4  @region(unreliable)
5  write(uint dest, uint src)
6     ensures (dest == src);
7
8  @region(unreliable)
9  write(uint dest, uint src)
10    when (!reliable)
11    ensures (err_min < dest);
```

Fig. 17. Switchable Rowhammer Model

Although researchers have devised protections from Rowhammer attacks [Kim et al. 2015, 2014], these protections are neither fully deployed (e.g., the DDR4 standard [Association et al. 2012] does not include protections) nor complete. For example, one commodity protection scheme is to use ECC memory that can detect and correct single-bit errors. While this protects the application from single-bit errors, double-bit errors (which are detectable but not correctable) and multi-bit (which are neither detectable or correctable) are possible as well [Kim et al. 2014].

Of the variety of possible Leto execution specifications for Rowhammer, the presented execution model captures a ECC scheme that for double-bit error stores a large value – of at least `err_min`

– into the memory location to communicate that an error happened. Such a model provides a hardware scheme that avoids the complexity of software-delivered ECC exceptions (which are not delivered at fine enough granularity to support recovery in practice [Lanteigne 2016]). This model can also simulate scenarios that selectively use ECC-protected caches [Alameldeen et al. 2011; Kim et al. 2007; Yoon and Erez 2009] in conjunction with traditional caches.

Switchability. The `reliable` flag models the fact that it is possible to selectively enable Rowhammer protection techniques [Aweke et al. 2016] (as well as ECC-protected caches) to trade performance for reliability. Specifically if `reliable` is set to `true`, then the model does not generate errors. In Figure 16, the program directly updates the value of `reliable` to model when it has enabled or disabled the protection scheme. In practice, this write would correspond to API calls that configure the underlying hardware (e.g., manipulations of memory-mapped registers).

Implementation. Connected components consists of two regions: the unreliable computation on Lines 6 through 12 and the reliable detection and correction code on Lines 15 through 23. The implementation may encounter errors on writes to the `next` vector because it has been allocated in the `relaxed` memory region. The implementation toggles the reliability of writes to `next` on Line 5 and Line 14.

The assertion that the developer verifies is that the algorithm correctly detects all errors, correctly recomputes errant values, and produces the same result as a reliable execution. This property is true if the choice of `min_error` in the execution model is large enough such that the condition on Line 16 is true if there was an error.

Detection. To verify that this SC-CC sketch detects all errors, the developer must verify that each errant value `next[i]` is greater than `i` (and therefore each errant value violates the nonincreasing property of the computation), which therefore triggers the reliable recomputation on Line 16. In SC-CC, the developer accomplishes this by assuming that that `min_err` is greater than `N` (the size of the graph, Line 1) – because graphs are dynamic in size – and, therefore, by definition, an errant value of `next[i]` is greater than `i` by being an invalid index.

Correction. Given that the detection phase soundly detects an error, the correction code on Line 19 reliably recomputes the value. The verification here must verify that the recomputation is functionally the same as the original computation performed by a reliable execution.

Summary. In the the final step, SC-CC asserts that `CC` is equivalent between both the original and relaxed executions, which follows from the verification of the correction step above (Line 25).

This implementation is an effective partition of reliability because the write on Line 9 may write to each `next[i]` up to `N` times. In the reliable loop, however, we can detect if an incorrect `write` happened by using a single read of the computed value of `next[i]`, a single recomputation of the result if it is incorrect, and then a reliable write to `next[i]`. This approach amortizes the overhead of checking each individual unreliable write to `next[i]` and – if the probability of an error is low – it is faster than an alternative implementation that stores `next[i]` reliably.

7 CASE STUDIES

We next present our results from using Leto to implement and verify several self-stabilizing and self-correcting algorithms. Figure 18 presents for each benchmark (Column 1) the execution model we verified under (Column 2), and the number of lines of code it contains (Column 3). Benchmarks with multiple models denote instances in which Leto can verify the application with a different model without the developer changing the program or its invariants.

Jacobi Iterative Method. We verify the Jacobi benchmark as presented in Section 2 under a multiplicative (SEU) error model (Figure 6).

Benchmark	Execution Model	LOC	Manual Annotations	Invariants Inferred
Jacobi	Multiplicative SEU	51	16	30
SS-CG	Additive SEU	163	22	36
SS-SD	Unbounded SEU	57	9	0
	Additive SEU			
	Multiplicative SEU			
SC-CC	Switchable Rowhammer	89	41	47
	Multicycle			

Fig. 18. Benchmark Verification Effort

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 the program state in the presence of faults [Sao and Vuduc 2013]. We verify under an additive SEU error model [Boston et al. 2018] that errors are sufficiently small such that the algorithm does not diverge. We also verify that the correction step can be correctly implemented using instruction duplication. We present a full description of the SS-CG our companion technical report [Boston et al. 2018].

Self-Stabilizing Steepest Descent (SS-SD). SS-SD is another iterative linear system of equations solver that employs a periodic, reliable correction step [Sao and Vuduc 2013] to repair the state of the program. We verify that a developer can correctly implement the correction step using instruction duplication (i.e., dual modular redundancy) under an unbounded SEU execution model (Figure 1), the additive SEU model [Boston et al. 2018], and the multiplicative SEU model. We present the full benchmark in our companion technical report [Boston et al. 2018].

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 initial computation step followed by a correction step [Sao et al. 2016] (Section 6). We verify that each iteration computes the correct result under a Rowhammer [Kim et al. 2014] error model that allows for an unbounded number of faulty writes to storage as well as a multicyle error model modified from that in Appendix A to have the appropriate error bounds. We specifically verify that the implementation detects and corrects all errors.

7.1 Verification Effort

Figure 18 also presents the annotation burden Leto imposes on the programmer. For each benchmark, we present the number of manual annotations (Column 4) and the number of automatically inferred loop invariants (Column 5). Manual annotations include loop invariants, assertions, and function requirements. We consider each conjunct a separate annotation when counting inferred invariants and manual annotations.

Results. We significantly reduce the number of invariants we must provide using inference in all but one benchmark. In half of the cases we infer more invariants than we provide. We infer no invariants for SS-SD as Z3 very quickly runs out of memory on our machine and therefore we must disable inference on all loops in that benchmark. We note that this failure results from the use of Z3's undecidable theories (Section 5). We believe that we could resolve this issue by monitoring the memory usage of the Z3 subprocess, killing the process if it consumes too much, and falling back to a weaker version of our invariant inference.

Runtime Characteristics. Figure 19 presents the runtime performance characteristics of the Leto C++ implementation. We ran our experiments on an Intel i5-5200U CPU clocked at 2.20GHz with 8 GB of RAM. For each benchmark we present the execution model we verified it under (Column 2), the time it took to run in seconds (Column 3), the maximum memory usage in kilobytes (Column 4), and the number of constraints generated for use with Z3 (Column 5).

Benchmark	Execution Model	Time (s)	Memory Usage (kbytes)	Constraints Generated
Jacobi	Multiplicative SEU	37.79	36132	12473
SS-CG	Additive SEU	4.29	37876	11665
SS-SD*	Unbounded SEU	0.19	25440	420
	Additive SEU	0.13	25452	420
	Multiplicative SEU	0.13	25664	420
SC-CC	Switchable Rowhammer	168.78	369556	4321
	Multicycle	174.75	405632	4168

Fig. 19. Benchmark Runtime Characteristics. *Memory consumption with invariant inference disabled.

8 RELATED WORK

Researchers have developed programming systems that enable developers to reason about *approximate computations*: computations for which the underlying execution substrate (e.g., the programming and/or hardware system) augments the application's behavior to produce approximate results [Carbin et al. 2013a; Hoffman et al. 2011; Misailovic et al. 2011, 2010; Rinard 2006].

For example, EnerJ [Sampson et al. 2011] and FlexJava [Park et al. 2015] enable developers to demonstrate non-interference between approximate computations and critical parts of the computation that should not be modified. Meola and Walker [2010] propose a sub-structural logic for reasoning about fault tolerant programs. Their logic enables the proof system to count the number of faults that have occurred and therefore reason about properties that may hold for one model but not another. Rely [Carbin et al. 2013b], Chisel [Misailovic et al. 2014], and Decaf [Boston et al. 2015], enable developers to reason about the *reliability* of their applications: the probability that they produce the correct result. In contrast to all of these approaches, Leto provides a more expressive and unconstrained logic that supports verifying complicated relational properties.

The work on the relaxed programming model [Carbin et al. 2012] provides developers with a Coq library to prove both safety and accuracy properties for relaxed computations by hand. Leto automates the relational logic of Carbin et al. [2012]. In principle, one can map Leto's integration of a hardware execution model to automatically instrumenting a program with the `relax` statements in the language of the relaxed programming model. A key difference in the formalism is that Leto lowers the logic down to a load-store machine instead of the high-level language of IMP. This formalism gives an explicit low-level semantics to memory and, for in-order processors, gives the first semantics and specifications for multicycle errors (Appendix A). While these results are not necessarily fully revolutionary, they provide valuable foundations for researchers in this space.

As to practicality, the manual proof method in Carbin et al. [2012] results in proofs on the order of hundreds of lines of Coq code for programs that are no more than 20 lines of code. This stands in contrast to Leto, in which the programs are significantly larger (a minimum of 2.5x larger) and have at most 38 annotations. Leto's base design (automatically weaving in the execution model), base automation, and loop invariant inference work together to make verification simpler.

He et al. [He et al. 2016, 2018] leverage the Symdiff framework [Lahiri et al. 2012] to automatically verify instances of approximate programs using the reasoning of Carbin et al. [2012]. However, these properties are simpler than the self-stability we verify here. However, still, in principle, it should be possible to translate instances of the Leto verification problem to Boogie programs and attempt to discharge them with Symdiff. One additional departure from a direct translation to Symdiff is that Leto's loop invariant inference approach incorporates labeled program points and model-based versioning (Section 5) to exploit the fact that the explicit model is the source of the deviation in the two semantics of the program. Specifically, Leto automatically reasons about model state changes between program points in an application (i.e., control flow) and automatically attempts to infer and propagate invariants between those points.

Chaudhuri et al. [2010, 2011] demonstrate analyses that enable reasoning about uncertain and approximate computations. These analyses specifically prove that either a program is continuous or robust with, in the latter case, robustness meaning that changes in the input to a computation result in linearly bounded changes in its outputs. Their results show that, for example, a variant of loop perforation, an approximate computing technique, can be soundly applied to programs that are robust. Robustness analysis presents a compelling, complementary approach for automating numerical analysis as well as viewing the soundness of an approximately transformed program as a judgment about the behavior of the original program on a perturbation to its input. Leto can similarly be used to bound the difference in a computation's outputs given perturbations to its inputs, such as the approximate vector-vector product in Section 2. However, a key distinction in this space is that verifying hardware fault tolerance includes, for example, verifying the correctness of dual modular redundancy, which is independent from robustness in the general case.

Relational Hoare Logic. Researchers have proposed relational Hoare Logics and verification systems to support verifying relational properties of programs [Barthe et al. 2011; Benton 2004; Carbin et al. 2012; Lahiri et al. 2012; Sousa and Dillig 2016]. The verification algorithms produced by Sousa and Dillig [Sousa and Dillig 2016] (Cartesian Hoare Logic) and Lahiri et al. [Lahiri et al. 2012] (Symdiff) demonstrate that it is possible to automatically compose proofs for relational verification. Leto's verification system differs from that of Cartesian Hoare Logic in that 1) the semantics of the two program executions are asymmetric and 2) 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 and therefore `assert` and `assumes` can often be matched to enable maximum reuse of assumed properties of the reliable execution during the verification of the relaxed.

Type Systems for Self-Stabilization. Self-Stabilizing Java provides developers with a type system and analysis that enables a developer to prove that any corrupted state of the program exits the system in a finite amount of time [Eom and Demsky 2012]. Leto's 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 to verify that the corruption in the program's state is small enough that the algorithm's correction steps will work as designed.

Fault Rate Analysis. Soft fault rates have led major organizations – such as Intel [Kurd et al. 2010; Mitra et al. 2005, 2006; Mukherjee et al. 2003], Google [Yim 2014], NASA [Johnston 2000], DOE [Snir et al. 2014], and DARPA [Amarasinghe et al. 2009] – to express concern over such faults.

The assumption of instruction-level arithmetic errors is the most common model for building 1) application-specific fault analyses and mechanisms [Bronevetsky and de Supinski 2008; Hoemmen and Heroux 2011; Huang and Abraham 1984; Oboril et al. 2011; Roy-Chowdhury and Banerjee 1994, 1996; Sao et al. 2016; Sao and Vuduc 2013; Shantharam et al. 2012], 2) software-level fault tolerance analyses and mechanisms [Li et al. 2016; Reis et al. 2005; Santini et al. 2017; Wei and Pattabiraman 2012; Yim et al. 2011], 3) micro-architectural resilience analyses and mechanisms [Austin 1999; Lu 1982; Meixner et al. 2007], and 4) circuit-level resilience analyses/mechanisms [Bowman et al. 2009, 2011; Kelin et al. 2010; Lilja et al. 2013; Quinn et al. 2015a,b; Turowski et al. 2015].

Mitra et al. found that combinational logic faults account for 11% of all soft errors [Mitra et al. 2005]. In addition, soft error rates, including combinational faults, are expected to increase as chips continue grow in the number of transistors [Mitra et al. 2005; Shivakumar et al. 2002]. These trends have inspired research on modeling the propagation of transient faults [Chen and Tahoori 2012; Omana et al. 2003], analyzing the rate of combinational soft faults [Buchner et al. 1997; Rao et al. 2007; Wang and Xie 2011; Zhang and Shanbhag 2006], analyzing the impact of combinational soft faults [Rajaraman et al. 2006], and correcting combinational logic faults [Mitra et al. 2006].

9 CONCLUSION

Emerging computational platforms are increasingly vulnerable to hardware errors. Future computations designed to execute on these platforms must therefore be designed to be *fault tolerant* and naturally resilient to error. We 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. Leto's support for execution models permit developers to convey information about the class of faults they expect their computational platforms to experience.

By giving developers tools for building verified, fault tolerant applications, Leto holds out the opportunity to develop new approximate, uncertain, and unreliable hardware with the confidence that the resulting systems can be reliably programmed.

A MULTICYCLE ERROR MODEL

```
 1  const real eps = ...;
 2  bool stuck = false;
 3  uint length;
 4
 5  operator*(real x1, real x2)
 6     when !stuck || length == 0
 7     ensures result == x1 * x2;
 8
 9  operator*(real x1, real x2)
10     when length > 0
11     modifies (stuck, length)
12     ensures stuck &&
13        length == old(length) - 1
```

Fig. 20. Multicycle Error Execution Model

Figure 20 presents a multicycle error execution model. A multicycle error is an error state in which multiple consecutive instructions experience errors [Inoue et al. 2011]. This implementation permits a single multicycle error and tracks the state of this error through the use of model variables `stuck` and `length`. The `stuck` flag represents whether or not the system is currently experiencing a multicycle fault while the `length` variable indicates how many instructions the fault will continue for. We leave the `length` variable unbound, permitting the multicycle error to persist for an arbitrary number of operations. Line 5 describes a reliable multiplication implementation that the model may use before the fault occurs (`!stuck`) and after it ends (`length == 0`). Line 9 encodes an operator that the model may use during, or to begin a multicycle error. The model may substitute these operators so long as a multicycle error has not occurred and resolved before the current instruction (`length > 0`). This operator sets `stuck`, decrements `length`, and constrains `result` to be within `eps` of the original result.

Together, these two operators ensure that at some point the system may be stuck experiencing faults on all multiplications, but after `length` multiplications the execution will be reliable.

ACKNOWLEDGMENTS

We thank Sara Achour, Clement Pit-Claudel, Benjamin Sherman, Eric Atkinson, Cambridge Yang, Deokhwan Kim, and the anonymous referees for their helpful comments on the paper.

This research was supported in part by the Department of Energy (DE-SC0008923 and DE-SC0014204) and the National Science Foundation (NSF CCF-1751011).

REFERENCES

Alaa R. Alameldeen, Ilya Wagner, Zeshan Chishti, Wei Wu, Chris Wilkerson, and Shih-Lien Lu. 2011. Energy-efficient Cache Design Using Variable-strength Error-correcting Codes *(ISCA)*.

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. 2009. ExaScale Software Study: Software Challenges in Extreme Scale Systems.

JEDEC Solid State Technology Association et al. 2012. JEDEC Standard: DDR4 SDRAM. *JESD79-4, Sep* (2012).

Todd M Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design *(MICRO)*.

Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-based protection against next-generation rowhammer attacks *(ASPLOS)*.

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs *(FMCO)*.

Mike Barnett, K Rustan M Leino, and Wolfram Schulte. 2004. The Spec# programming system: An overview *(CASSIS)*.

G. Barthe, J. Crespo, and C. Kunz. 2011. Relational verification using product programs *(FM)*.

N. Benton. 2004. Simple relational correctness proofs for static analyses and program transformations *(POPL)*.

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

Brett Boston, Zoe Gong, and Michael Carbin. 2018. Verifying Programs Under Custom Application-Specific Execution Models *(arXiv 1805.06090)*.

Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability type inference for flexible approximate programming *(OOPSLA)*.

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

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. 2011. A 45 nm resilient microprocessor core for dynamic variation tolerance. *IEEE Journal of Solid-State Circuits* 46, 1 (2011), 194–208.

Greg Bronevetsky and Bronis de Supinski. 2008. Soft error vulnerability of iterative linear algebra methods *(ICS)*.

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

M. Carbin, D. Kim, S. Misailovic, and M. Rinard. 2012. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs *(PLDI)*.

M. Carbin, D. Kim, S. Misailovic, and M. Rinard. 2013a. Verified integrity properties for safe approximate program transformations *(PEPM)*.

M. Carbin, S. Misailovic, and M. Rinard. 2013b. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware *(OOPSLA)*.

Michael Carbin and Martin C. Rinard. 2010. Automatically Identifying Critical Input Regions and Code in Applications *(ISSTA)*.

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity Analysis of Programs *(POPL)*.

Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. 2011. Proving Programs Robust *(ESEC/FSE)*.

Daniel Chen, Gabriela Jacques-Silva, Zbigniew Kalbarczyk, Ravishankar K Iyer, and Bruce Mealey. 2008. Error behavior comparison of multiple computing systems: A case study using Linux on Pentium, Solaris on SPARC, and AIX on POWER *(PRDC)*.

Liang Chen and Mehdi B Tahoori. 2012. An efficient probability framework for error propagation and correlation estimation *(IOLTS)*.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver *(TACAS)*.

Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. 2012. Algorithm-based Fault Tolerance for Dense Matrix Factorizations *(PPoPP)*.

Yong hun Eom and Brian Demsky. 2012. Self-stabilizing Java *(PLDI)*.

Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an annotation assistant for ESC/Java *(FME)*.

Carlo Alberto Furia and Bertrand Meyer. 2010. Fields of Logic and Computation. Springer-Verlag, Chapter Inferring Loop Invariants Using Postconditions, 277–300.

Shaobo He, Shuvendu K Lahiri, and Zvonimir Rakamarić. 2016. Verifying relative safety, accuracy, and termination for program approximations *(NFM)*.

Shaobo He, Shuvendu K. Lahiri, and Zvonimir Rakamarić. 2018. Verifying Relative Safety, Accuracy, and Termination for Program Approximations. *Journal of Automated Reasoning* 60, 1 (2018).

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

Mark Hoemmen and Michael A Heroux. 2011. Fault-tolerant iterative methods via selective reliability *(SC)*.

H. Hoffman, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. 2011. Dynamic Knobs for Responsive Power-Aware Computing *(ASPLOS)*.

Kuang-Hua Huang and Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6.

Tomoo Inoue, Hayato Henmi, Yuki Yoshikawa, and Hideyuki Ichihara. 2011. High-level synthesis for multi-cycle transient fault tolerant datapaths *(IOLTS)*.

C. G. J. Jacobi. 1845. Ueber eine neue AuflÃűsungsart der bei der Methode der kleinsten Quadrate vorkommenden lineÃďren Gleichungen. *Astronomische Nachrichten* 22, 20 (1845), 297–306.

Allan H Johnston. 2000. Scaling and technology issues for soft error rates. (2000).

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

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

Jangwoo Kim, Nikos Hardavellas, Ken Mai, Babak Falsafi, and James Hoe. 2007. Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding *(MICRO)*.

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

Nasser A Kurd, Subramani Bhamidipati, Christopher Mozak, Jeffrey L Miller, Timothy M Wilson, Mahadev Nemani, and Muntaquim Chowdhury. 2010. Westmere: A family of 32nm IA processors *(ISSCC)*.

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs *(CAV)*.

Mark Lanteigne. 2016. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware.

Tuo Li, Jude Angelo Ambrose, Roshan Ragel, and Sri Parameswaran. 2016. Processor Design for Soft Errors: Challenges and State of the Art. *ACM Computing Surveys (CSUR)* 49, 3 (2016), 57.

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

David J. Lu. 1982. Watchdog processors and structural integrity checking. *IEEE Trans. Comput.* 31, 7 (1982), 681–685.

Albert Meixner, Michael E Bauer, and Daniel Sorin. 2007. Argus: Low-cost, comprehensive error detection in simple cores *(MICRO)*.

Matthew L. Meola and David Walker. 2010. Faulty Logic: Reasoning About Fault Tolerant Programs *(ESOP)*.

Bertrand Meyer. 1992. *Eiffel: The Language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

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

S. Misailovic, D. Roy, and M. Rinard. 2011. Probabilistically Accurate Program Transformations *(SAS)*.

S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. 2010. Quality of service profiling *(ICSE)*.

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

Subhasish Mitra, Ming Zhang, Saad Waqas, Norbert Seifert, Balkaran Gill, and Kee Sup Kim. 2006. Combinational logic soft error correction *(ESOP)*.

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

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

Martin Omana, Giacinto Papasso, Daniele Rossi, and Cecilia Metra. 2003. A model for transient fault propagation in combinatorial logic *(IOLTS)*.

Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. FlexJava: Language Support for Safe and Modular Approximate Programming *(FSE)*.

RC Quinn, JS Kauppila, TD Loveless, JA Maharrey, JD Rowe, ML Alles, BL Bhuva, RA Reed, M Mounasser, K Lilja, and LW Massengill. 2015a. Frequency Trends Observed in 32nm SOI Flip-Flops and Combinational Logic. *IEEE Transactions on Nuclear Science* (2015).

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. 2015b. Heavy ion SEU test data for 32nm SOI flip-flops *(REDW)*.

R Rajaraman, JS Kim, Narayanan Vijaykrishnan, Yuan Xie, and Mary Jane Irwin. 2006. SEAT-LA: A soft error analysis tool for combinational logic *(VLSI Design)*.

Rajeev R Rao, Kaviraj Chopra, David T Blaauw, and Dennis M Sylvester. 2007. 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 (2007), 468–479.

G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. 2005. SWIFT: Software Implemented Fault Tolerance *(CGO)*.

M. Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks *(ICS)*.

Amber Roy-Chowdhury and Prithviraj Banerjee. 1994. Algorithm-based fault location and recovery for matrix computations *(FTCS)*.

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

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation *(PLDI)*.

Thiago Santini, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. 2017. Effectiveness of Software-Based Hardening for Radiation-Induced Soft Errors in Real-Time Operating Systems *(ARCS)*.

Piyush Sao, Oded Green, Chirag Jain, and Richard Vuduc. 2016. A Self-Correcting Connected Components Algorithm *(FTXS)*.

Piyush Sao and Richard Vuduc. 2013. Self-stabilizing iterative solvers *(ScalA)*.

Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. 2012. Fault tolerant preconditioned conjugate gradient for sparse linear system solution *(ICS)*.

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

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. 2014. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications* 28, 2 (2014), 129–173.

M. Sousa and I. Dillig. 2016. Cartesian Hoare Logic for Verifying K-safety Properties *(PLDI)*.

Michael B Sullivan and Earl E Swartzlander. 2012. Truncated error correction for flexible approximate multiplication *(ASILOMAR)*.

Michael B Sullivan and Earl E Swartzlander. 2013. Truncated logarithmic approximation *(ARITH)*.

Anna Thomas and Karthik Pattabiraman. 2016. Error Detector Placement for Soft Computing Applications. *ACM Trans. Embed. Comput. Syst.* (2016).

M Turowski, K Lilja, K Rodbell, and P Oldiges. 2015. 32nm SOI SRAM and latch SEU crosssections measured (heavy ion data) and determined with simulations *(SEE)*.

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

Sriram Krishnamoorthy Vishal Chandra Sharma, Ganesh Gopalakrishnan. 2016. Towards Resiliency Evaluation of Vector Programs *(DPDNS)*.

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

Jiesheng Wei and Karthik Pattabiraman. 2012. BLOCKWATCH: Leveraging similarity in parallel programs for error detection *(DSN)*.

Keun Soo Yim. 2014. Characterization of impact of transient faults and detection of data corruption errors in large-scale n-body programs using graphics processing units *(IPDPS)*.

Keun Soo Yim, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2010. Measurement-based analysis of fault and error sensitivities of dynamic memory *(DSN)*.

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

Doe Hyun Yoon and Mattan Erez. 2009. Memory Mapped ECC: Low-cost Error Protection for Last Level Caches *(ISCA)*.

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