SymPLFIED: Symbolic Program-Level Fault Injection and Error Detection Framework

Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer
Coordinated Science Laboratory, University of Illinois at Urbana-Champaign
{pattabir, nakka, kalbarcz, rkiyer}@uiuc.edu

Abstract

This paper introduces SymPLFIED, a program-level framework that allows specification of arbitrary error detectors and the verification of their efficacy against hardware errors. SymPLFIED comprehensively enumerates all transient hardware errors in registers, memory, and computation (expressed as value errors) that potentially evade detection and cause program failure. The framework uses symbolic execution to abstract the state of erroneous values in the program and model checking to comprehensively find all errors that evade detection. We demonstrate the use of SymPLFIED on a widely deployed aircraft collision avoidance application, tcas. *Our results show that the SymPLFIED framework can be used to uncover hard-to-detect corner cases caused by transient errors in programs that may not be exposed by random fault-injection based validation.*

Keywords: Dependability validation, Fault injection, Symbolic execution, Error detectors, Model checking.

1 Introduction

Error detection mechanisms are vital for building highly reliable systems. There has been significant work on efficiently placing and deriving error detectors for programs [1]. An important challenge is to enumerate the set of errors the mechanism fails to detect, from either a known set or an unknown set. Typically, verification techniques target the defined set of errors the detector is supposed to detect. While this is valuable, one cannot predict the kinds of errors that may occur in the field, and hence it is important to evaluate detectors under arbitrary conditions.

Fault -injection is a well-established technique to evaluate the coverage of error detection mechanisms [2]. However, there is a compelling need to develop a formal framework to reason about the efficiency of error detectors as a complement to traditional fault injection. This can uncover possible "corner cases," which may be missed by conventional fault injection due to its inherent statistical nature.

This paper presents SymPLFIED, a framework for verifying error detectors in programs using symbolic execution and model checking. The goal of the framework is to expose error cases that would potentially escape detection and cause program failure. The focus is on transient hardware errors. The framework makes the following unique contributions:

1. Introduces a formal model to represent programs expressed in a generic assembly language, and reasons about the effects of errors originating in hardware and propagating to the application without assuming specific detection mechanisms;
2. Specifies the semantics of general error detectors using the same formalism, which allows verification of their detection capabilities;
3. Represents errors using a single symbol, thereby coalescing multiple error values into a single symbolic value in the program. This includes single- and multi-bit errors in the register file, main memory, cache, as well as errors in computation.

To the best of our knowledge, this is the first framework that models the effect of arbitrary hardware errors on software, independent of the underlying detection mechanism. It uses model checking to exhaustively enumerate the consequences of the symbolic errors on the program. The analysis is completely automated and does not miss errors that might occur in a real execution. However, as a result of symbolically abstracting erroneous values, it may discover errors that may not manifest in the real execution of the program, i.e., false positives.

To evaluate the framework, the effects of hardware transient errors were considered on a commercially deployed application, *tcas*. The framework identified errors that lead to a catastrophic outcome in the application, errors not found by statistical fault-injection in a comparable amount of time. The frame-

work was also demonstrated on a larger program, *replace*, to demonstrate its scalability.

2 Related Work

Formal techniques have been extensively applied to **hardware verification** [3]. Hardware verification techniques typically focus on unmasking hardware design defects as opposed to transient errors due to electrical disturbances or radiation.

Formal techniques have been also used in **software verification**, i.e., to prove that a program's code satisfies a programmer-supplied specification [4]. Typically, program verification techniques are geared toward finding software defects, and they assume that the hardware and the program environment are error-free. In other words, they prove that the program satisfies the specification *provided that* the hardware platform on which the program is executed does not experience errors.

The techniques presented in [5] and [6] consider the effects of hardware transient errors (*soft errors*) on programs implemented in hardware. While these techniques are useful for applications implemented as hardware circuits, it is not clear how the technique can be extended for reasoning about the effects of errors on programs. This is because programs are normally executed on general-purpose processors in which the manifestation of a low-level error is different from that of an error in a circuit implementing the application.

Many error detection mechanisms have been proposed in the literature, along with formal proofs of their correctness. However, the verification methodology is usually tightly coupled with the mechanism under study. For example, [7] proposes and verifies a control-flow checking technique by constructing a hypothetical program augmented with the technique and model checking the program for missed detections. The program is carefully constructed to exercise all possible cases of the control-flow checking technique. It is non-trivial to construct such programs for each error detection mechanism to be verified.

A recent paper [8] proposes the use of type-checking to verify the fault-tolerance provided by a specific error detection mechanism, namely, compiler-based instruction duplication. The paper proposes a detailed machine model for executing programs. The faults in the fault model (single-event upsets) are represented as transitions in this machine model. The advantage of the technique is that it allows reasoning about the effect of low-level hardware faults on the whole program rather than on individual instructions or data. However, the detection mechanism (duplication) is tightly coupled with the machine model, due to inherent assumptions that limit error propagation in the program and may not hold in non-duplicated programs. Further, the type-checking technique in [8] either accepts or rejects a program based on whether the program has been duplicated correctly, but it does not consider the consequences of the error on the program. As a result, the program may be rejected by the technique even though the error is benign and has no effect on the program's output.

Symbolic execution has been used for a wide variety of software testing and maintenance purposes [9]. The main idea in these techniques is to execute the program with symbolic values rather than concrete values and to abstract the program state as symbolic expressions. An example of a commercially deployed symbolic execution technique to find bugs in programs is Prefix [10]. However, like many symbolic execution techniques, Prefix assumes that the hardware does not experience errors during program execution.

Recently, [11] introduced a symbolic approach for injecting faults into programs. The goals of this approach are similar to ours, namely to verify properties of fault-tolerance mechanisms in the presence of hardware errors. The technique reasons on programs written in Java and considers the effect of bit-flips in program variables. However, a hardware error can have wide-ranging consequences on the program, including changing its control-flow and affecting the runtime support mechanisms for the language (such as the program stack and libraries). These errors are not considered by the technique. Further, the technique presented in [11] uses theorem proving to verify the error-resilience of programs. Theorem proving has the intrinsic advantage that it is naturally symbolic and can reason about the non-determinism introduced by errors. However, as it stands today, theorem proving requires considerable programmer intervention and expertise, and cannot be completely automated.

Summary: The formal techniques considered in this section predominantly fall into the category of software-only techniques which do not consider hardware errors [4], or into the category of hardware-only techniques which do not consider the effects of errors on software [3]. Further, existing verification techniques are often coupled with the detection mechanism (e.g. duplication) being verified [7][8]. Therefore, there exists no *generic* technique that allows reasoning about the effects of *arbitrary* hardware faults on software, and can be combined with an arbitrary fault model and detection technique(s). This is important for enumerating all hardware transient errors that would escape detection and cause programs to fail.

3 Approach

This section introduces the conceptual model of the SymPLFIED framework and also the technique used by SymPLFIED to symbolically propagate errors in the program. The fault-model used by SymPLFIED is also discussed.

3.1 Framework

The SymPLFIED framework accepts a program protected with error detectors and enumerates all errors (in a particular class) that would not be detected by the detectors in the program. Figure 1 presents the conceptual design of the SymPLFIED framework.

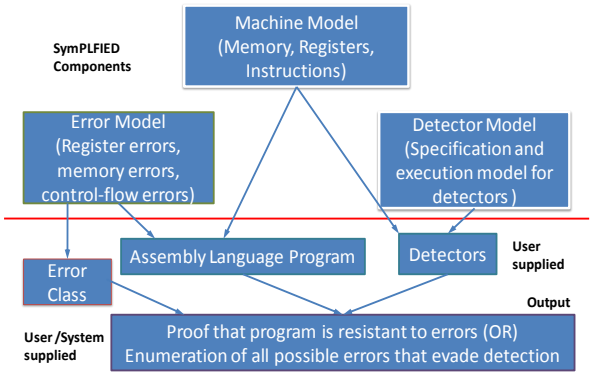

Figure 1: Conceptual design flow of SymPLFIED

Assembly Language: The SymPLFIED framework considers programs represented at the assembly language level. The advantage of using assembly language is that many low-level hardware errors that impact the program can be represented at the assembly language level (as shown in section 3.3). Further, the entire application, including runtime libraries, is amenable to analysis at the assembly language level. While it may be argued that we need to go to even lower levels (e.g., register transfer level) to truly represent low-level errors, the consequent state explosion can impact the practicality of the model.

We define a generic assembly language in which programs are represented for formal analysis by the framework. Because the language defines a set of architectural abstractions found in many common Reduced Instruction Set Computing (RISC) architectures, it is portable across these architectures, with an architecture-specific front end. The assembly language has direct support for (1) input/output operations, so that programs can be analyzed independently of the operating system, and (2) invocation of error detectors using special instructions, called *CHECK*s, which allow detectors to be represented in line with the program.

Inputs: The inputs to the framework are (1) a program written in the generic assembly language (discussed above), (2) error detectors embedded in the program code, and (3) a class of hardware errors to be considered as part of the fault model (e.g. control-flow errors, register file errors).

Operation: The program expressed in the generic assembly language is automatically translated into a formal mathematical model (represented using the Maude system [12]). Since the abstraction is close to the actual program in assembly language, it is sufficient for the user to formulate generic specifications, such as an incorrect program outcome or an exception being thrown. Such a low-level abstraction of the program is useful to reason about hardware errors. The formal model can then be rigorously analyzed under error conditions against the above specifications using techniques such as model checking and theorem proving. In this paper, model checking is used because it is completely automated and requires no intervention on the part of the programmer.

Outputs: The framework uses the symbolic fault-propagation technique (section 3.2) and outputs either:

1. A proof that the program with the embedded detectors is resilient to the error class considered, or
2. A comprehensive set of all errors belonging to the error class that may evade detection and lead to program failure (crash, hang, incorrect output).

Components: The framework consists of the following formal models:

- **Machine Model:** Models the formal semantics of the machine on which the program is to be executed (e.g. registers, memory, instructions, etc.).
- **Error Model:** Specifies error classes and error manifestations in the machine on which the program is executed, e.g., errors in the class *register errors* can manifest in any register in the machine.
- **Detector Model:** Specifies the format of error detectors and their execution semantics. It also includes the action taken upon detecting the error, e.g., halting the program.

By representing all three models in the same formal framework, it is possible to reason about the effects of errors (in the error model) both on programs (represented in the machine model) and on detectors (represented in the detector model) in a unified way.

Correctness: For the results of the formal analysis to be trustworthy, the model must be provably correct. There are two aspects to correctness, namely:

1. The model must satisfy certain desirable properties, such as termination, coherence, and sufficient completeness [12]. We have formally analyzed the specification using automated checking tools available in Maude to ensure these properties.
2. The model must be an accurate representation of the system being modeled. We have also validated the model by rigorously analyzing the behavior of errors in the model and comparing it with the behavior of the real system.

3.2 Symbolic Fault Propagation

The SymPLFIED approach represents the state of all erroneous values in the program using the abstract symbol *err*. The *err* symbol is propagated to different locations in the program during execution using simple error propagation rules (shown in section 5.2). The

symbol also introduces non-determinism in the program when used either in the context of comparison and branch instructions or as a pointer operand in memory operations. Because the same symbol is used to represent all erroneous values in the program, the approach distinguishes program states based on where errors occur rather than on the nature of the individual error(s). As a result, it avoids state explosion and can keep track of all possible places in the program the error may propagate to starting from its origin. However, because errors in data values are not distinguished from each other, the set of error states corresponding to a fault is over-approximated. This can result in the technique's finding erroneous program outcomes that may not occur in a real execution. For example, if an error propagates from a program variable *A* to another variable *B*, the variable *B*'s value is constrained by the value of the variable *A*. In other words, given a concrete value of *A* after it has been affected by the error, the value of *B* can be uniquely determined due to the error propagating from *A* to *B*. The SymPLFIED technique on the other hand, assigns a symbolic value of *err* to both variables, and would not capture the constraint on *B* due to the variable *A*. As a result, it would not be able to determine the value in register *B* even when given the value in register *A*. This may result in the technique discovering spurious program outcomes. Such spurious outcomes are termed *false positives*.

While SymPLFIED may uncover false positives, it will never miss an outcome that may occur in the program due to the error (in a real execution). This is because SymPLFIED systematically explores the space of all possible manifestations of the error on the program. Hence, the technique is *sound*, meaning it finds all error manifestations, but it is not always *accurate*, meaning that it may find false positives.

Our premise is that soundness is more important than accuracy from the point of view of designing detection mechanisms, as we can augment the set of error detectors to conservatively detect all the errors identified by the technique (including false positives).

While a small number of false positives can be tolerated, it must be ensured that the technique does not find too many false positives, as the cost of developing detectors to protect against false positives can overwhelm the benefits provided by detection. The SymPLFIED technique uses a custom constraint solver to remove false positives in the search space. This is similar to the technique used in [13].

3.3 Fault Model

The fault model considered by SymPLFIED includes transient errors in memory/registers and computation. Permanent or intermittent errors are not considered. Transient errors are modeled as follows:

Errors in memory/registers are modeled by replacing the contents of the memory location or register by the symbol *err*. No distinction is made between single- and multi-bit errors.

Errors in computation are modeled based on where they occur in the processor pipeline and how they affect the architectural state. Table 1 shows how such errors are modeled by the framework.

Errors in processor control logic (such as in the register renaming unit) are not considered. This is a topic for future work.

3.4 Scalability

As in most model checking approaches, the exhaustive search performed by SymPLFIED can be exponential in the number of instructions executed by the program (in the worst case). However, the error detection mechanisms in the program can be used to optimize the state space exploration process. For example, if a certain code component protected with detectors is proved to be resilient to all errors of a particular class, then such errors can be ignored when considering the space of errors that can occur in the system as a whole. This suggests a hierarchical or compositional approach, where first the detection mechanisms deployed in small components are proved to protect that component from errors of a particular class, and then inter-component interactions are considered. This is an area of future investigation.

4 Examples

This section illustrates the SymPLFIED approach in the context of an application that calculates the factorial of a number, shown in Figure 2. The program is represented in the generic assembly language discussed in Section 3.1.

4.1 Error Injection

We illustrate our approach with an example of an injected error in the program shown in Figure 1. Assume that a fault occurs in register *$3* (which holds the value of the loop counter variable) in line 8 of the program after the loop counter is decremented (*subi $3 $3 1*). The effect of the fault is to replace the contents of the register *$3* with *err*. The loop back-edge is then executed, and the loop condition is evaluated. Since *$3* has the value *err* in it, it cannot be uniquely determined whether the loop condition evaluates to true or false. Therefore, the execution is forked so that the loop condition evaluates to true in one case and to false in the other. The *true* case exits immediately and prints the value stored in *$2*. Since the error can occur in any loop iteration, the value printed can be any of the following: *1!, 2!, 3!, 4!, 5!*.

Table 1: Errors in pipeline stages and how they are modeled by the framework

Fault Origin	Error Symptom	Conditions under Which Modeled	Modeling Procedure	
Instruction Decoder	One of the fields of an instruction is corrupted.	One valid instruction is converted to another valid instruction.	Instructions writing to a destination (e.g., *add*) - change the output target	*err* in the original and new targets (register or memory)
			Instructions with no target (e.g., *nop*) – replace with instructions with targets (*e.g. add)*	*err* in the new wrong target (register or memory)
			Instructions with a single destination (e.g.*add*) – replace with instruction with no target (e.g. *nop*)	*err* in the original target location (register or memory)
Address or Data Bus	Data read from memory, cache, or register file is corrupted.	Single and multiple bit errors in the bus during instruction execution.	Errors in register data bus	*err* in source register(s) of the current instruction
			Error in cache bus	*err* in target registers of *load* instructions to the location
			Error in memory bus	*err* in target register of *load* instructions to the location
Processor Functional Unit	Functional unit output is corrupted.	Single and multiple bit errors in registers/memory.	Functional Unit output to register or memory	*err* in register or memory file being written to by the current instruction
Instruction Fetch Mechanism	Errors in the fetch unit.	Single or multiple bit errors in *PC* or instruction.	Fetch from an erroneous location due to error in *PC*	*PC* is changed to an arbitrary but valid code location
			Error in instruction while fetching	Modeled as errors in Instruction Decoder (described above).

```
1        ori  $2  $0  #1     --- initial product p = 1
2        read $1              --- read i from input
3        mov $3, $1
4        ori $4 $0 #1        --- for comparison  purposes
loop:    setgt $5 $3 $4      --- start of loop
6        beq  $5 0 exit      ---- loop condition : $3 > $4
7        mult $2 $2 $3            ---- p = p * i
8        subi $3 $3 #1            ---- i = i - 1
9        beq  $0 #0 loop          --- loop backedge
exit:    prints "Factorial = "
11         print $2
12         halt
```

Figure 2: Program to compute factorial

The *false* case continues executing the loop, and the *err* value is propagated from register *$3* to register *$2* due to the multiplication operation (*mul $2 $2 $3*). The program then executes the loop back-edge and evaluates the branch condition. Again, the condition cannot be resolved, as register $3 is still *err*. The execution is forked again, and the process is repeated *ad infinitum*. In practical terms, the loop is terminated after a certain number of instructions and the value *err* is printed; otherwise, the program times out (detected by a watchdog mechanism) and is stopped.

Complexity: For a physical fault-injection approach to discover the same set of outcomes for the program as SymPLFIED, it would need to inject all possible values (in the integer range) into the loop counter variable. This can correspond to 2^k cases in the worst case, where k is the number of bits used to represent an integer. In contrast, SymPLFIED considers at most $n+1$ possible cases in this example, where n is the number of iterations of the loop. This is because each fork of the execution at the loop condition results in the *true* case exiting the loop and the program. In the general case, SymPLFIED may need to consider 2^n possible cases. However, by upper-bounding the number of instructions executed in the program, the growth in the search space can be controlled (see section 5.4).

False positives: In the above example, not all errors in the loop counter variables will cause the loop to terminate early. For example, an error in the higher-order bits of the loop counter variable in register *$3* may still cause the loop condition (*$3 > $4*) to be *false*. However, SymPLFIED would assume that both the *true* and *false* cases are possible, as it does not distinguish among errors in different bit positions.

4.2 Error Detection

We now discuss how SymPLFIED supports error detection mechanisms in the program. Figure 3 shows the same program augmented with error detectors. Recall that detectors are invoked through special CHECK instructions, as explained in Section 3.1. The error detectors together with their supporting instructions (*mov* instruction in line 8) are shown in bold.

The same error is injected as before in register *$3* (the new line number is 11). As shown earlier, the loop back-edge is executed, and the execution is forked at the loop condition (*$3 > $4*).

The *true* case exits immediately, while the *false* case continues executing the loop. The *false* case "remembers" that the loop condition (*$3 < $4*) is false by adding this as a constraint to the search. The *false* case then encounters the first detector that checks if (*$4 < $3*). The check always evaluates to *true* because of the constraint and hence does not detect the error. The program continues execution, and the error propagates to *$2* in the *mul* instruction. However, the value of *$2* from the previous iteration does not have an error in it,

and this value is copied to register *$6* by the *mov* instruction in line 8. Therefore, when the second detector is encountered within the loop (line 10), the LHS of the check evaluates to *err* and the RHS evaluates to (*$6* × *$1*), which is an integer.

```
1        ori  $2  $0  #1              --- initial product p = 1
2        read $1                      --- read i from input
3        mov $3, $1
4        ori $4 $0 #1                 --- for comparison purposes
loop:  setgt $5 $3 $4                 --- start of loop
6          beq  $5 0 exit
7          check ($4 < $3)
8          mov $6, $2
9        mult $2 $2 $3          ---- p = p * i
10         check ($2 >= $6 * $1)
11       subi $3 $3 #1          ---- i = i - 1
12         beq  $0 #0 loop      --- loop backedge
exit:   prints "Factorial = "
14         print $2
15         halt
```

Figure 3: Factorial program with error detectors

The execution is forked once again at the second detector into *true* and *false* cases. The *true* case continues execution and propagates the error in the program as before. The *false* case of the check throws an exception, and the detector fails, thereby detecting the error. The constraints for the *false* case, namely, (*$6* × *$3* ≥ *$6* × *$1*) are also remembered. Based on this constraint, as well as the earlier constraint (*$3* > *$4*), the constraint-solver deduces that the second detector will detect the error if and only if the fault in register $3 causes it to have a value greater than the initial value read from the input (stored in register $1).

The programmer can then formulate a detector to handle the case when the error causes the value of register *$3* to be less than the original value in register *$1*. Therefore, the errors that evade detection are made explicit to the programmer (or to an automated mechanism) who can then make an informed decision about handling the errors.

The error considered above is only one of many possible errors that may evade detection in the program. These errors are too numerous for manual inspection and analysis as done in this example. Moreover, not all errors that evade detection lead to program failure (due to logical masking and dead values).

The main advantage of SymPLFIED is that it can quickly isolate the errors that would evade detection and cause program failure from the set of all possible transient errors that can occur in the program. It can also show an execution trace of how the error evaded detection and led to the failure. This is important in order to understand the weaknesses in existing detection mechanisms and improve them.

5 Implementation

We have implemented the SymPLFIED framework using the Maude rewriting logic system. Rewriting logic is a general-purpose logical framework for specification of programming languages and systems [12]. Maude is a high-performance reflective language and system that supports rewriting logic specification and programming for a wide range of applications [12]. Maude allows a wide variety of formal analysis techniques to be applied on the same specification.

5.1 Machine Model

This section describes the machine model for executing assembly language programs using Maude.

Equations and Rules: As far as possible, we have used equations instead of rewrite rules for specifying the models. The main advantage of using equations is that Maude performs rewriting using equations much faster than using rewrite rules. However, equations must be deterministic and cannot accommodate ambiguity. The machine model is completely deterministic because, for a given instruction sequence, the final state can be uniquely determined in the absence of errors. Therefore, the machine model can be represented entirely using equations. However, the error model is non-deterministic and requires rules.

Assumptions: The following assumptions are made by the machine model:

- An attempt to fetch an instruction from an invalid code address results in an "illegal instruction" exception being thrown. The set of valid addresses is defined at program load time by the loader.
- Memory locations are defined when they are first written to (by store instructions). An attempt to read from an undefined memory location results in an "illegal address" exception being thrown. It is assumed that the program loader initializes all locations prior to their first use in the program.
- Program instructions are assumed to be immutable and hence cannot be overwritten during execution or affected by data errors.
- Arithmetic operations are supported only on integers and not on floating point numbers.

Machine State: The central abstraction used in the machine model is the notion of *machine state*, which consists of the mutable components of the machine's structures. The machine state is carried from instruction to instruction in program execution order, with each instruction optionally looking up and/or updating the state's contents. For example, *PC(pc) regs(R) mem(M) input(In) output(out)* represents a machine state in which the current program counter is *pc*, register file is *R*, memory is *M*, and input and output streams are *in* and *out*, respectively.

Instruction Execution: We consider example instructions from two different instruction classes and illustrate the Maude equations used to model them.

These equations use the *fetch* primitive to retrieve instructions from memory, as well as the standard lookup and update operations of registers and memory. These details are skipped due to space constraints.

1. Arithmetic Instruction: Consider the execution of the *addi* instruction, which adds the value[1] *v* to the register given by *rs* and stores the result in register *rd*. In the equation given below, *C* represents the code, which is assumed to be immutable and hence is shown to be unchanged by the instruction execution. The <_,_> operator represents the new machine state obtained by executing an instruction (the first argument) on a machine state (the second argument).

eq { C, < addi rd rs v , PC(pc) regs(R) S > } = { C, < fetch(C, pc), PC(next(pc)) regs(R[rd] <- R[rs] + v) S > } .

The elements of the machine state in the above equations are composable and hence can be matched with a generic symbol *S* representing the rest of the state. This allows new machine-state elements to be added without modifying existing equations.

2. Branch Instructions: Consider the example of the *beq rs, v, l* instruction, which branches to the code label *l* if and only if the register *rs* contains the constant value *v*. The equation for *beq* is similar to the equation for the *addi* operation except that it uses the in-built if-then-else operator of Maude.

eq { C , < beq rs v l , pc(PC) regs(R) S > = if isEqual(R[rs], v) then { C, < fetch(C, pc), PC(next(pc)) regs(R) S >} else { C, < fetch(C, l), PC(l) regs(R) S >} fi .

Note the use of the *isEqual* primitive rather than a direct == to compare the values of the register *rs* and the constant value *v*. This is because the register *rs* may contain the symbolic constant *err* and hence needs to be resolved according to the error model.

Similarly, memory instructions, input/output operations, and special instructions such as *halt* and *throw* are modeled by the framework. The details are skipped due to space constraints, but may be found in the technical report version of the paper [14].

5.2 Error Model

The overall approach to error injection and propagation was discussed in Section 3.2. In this section we discuss the implementation of the approach using rewriting logic in Maude. The implementation of the error model is divided into the following sub-models.

Error Injection Sub-Model: The error-injection sub-model is responsible for introducing symbolic errors into the program during its execution. The injector can be used to introduce the *err* symbol into registers, memory locations, or the program counter when the program reaches a specific location in the code. This is implemented by adding a breakpoint mechanism to the machine model. The choice of the register or memory location to inject into is made non-deterministically by the injection sub-model using rewrite rules (not shown due to space reasons).

Error Propagation Sub-Model: Once an error has been injected, it is allowed to propagate through the equations for executing the program in the machine model. The rules for error propagation are described independently of the machine model using equations as follows:

eq err + err = err . eq err + I = err . eq I + err = err .
eq err – err = err . eq err – I = err . eq I – err = err .
*eq err * I = if (I==0) then 0 else err fi .*
*eq I * err = if (I==0) then 0 else err fi .*
eq err / I = if (I==0) then throw "div--zero" else err fi .
eq I / err = if isEqual(err, 0) then throw "div- zero" else err fi
*eq err * err = if isEqual(err, 0) then 0 else err fi .*
eq err / err = if isEqual(err, 0) then throw "div.." else err fi

In the equation above, *I* represents an integer. Note that any arithmetic operation involving *err* also evaluates to *err* (unless it is multiplied by 0). Note also how the divide-by-zero case is handled in the equation for the divide operation.

Comparison Handling Sub-Model: The rules for comparisons involving one or more *err* values are expressed as rewrite-rules, as they are non-deterministic in nature. For example, the rewrite rules for the *isEqual* operator are as follows:

rl isEqual(I, err) => true . rl isEqual(I, err) => false .
rl isEqual(err, err) => true . rl isEqual(err, err) => false .

The comparison operators involving *err* operands evaluate to either true or false non-deterministically. This is equivalent to forking the program's execution into the true and false cases. However, once the execution has been forked, the outcome of the comparison is deterministic and subsequent comparisons involving the same unmodified locations must return the same outcome (otherwise false positives will result). This can be accomplished by updating the state (after forking the execution) with the results of the comparison. This is handled by the *constraint solving* sub-model. More details may be found in the technical report [14].

Memory- and Control-Handling Sub-Model: Memory and control errors are handled using rewrite rules (because they are non-deterministic) as follows:

Errors in jump or branch targets: The program either jumps to an arbitrary (but valid) code location or throws an "illegal instruction" exception.

Errors in pointer values of loads: The program either retrieves the contents of an arbitrary memory location or throws an "illegal-address" exception.

Errors in pointer values of stores: The program either overwrites the contents of an arbitrary memory location or creates a new value in memory.

5.3 Detector Model

Error detectors are defined as executable checks in the program that test whether a given memory loca-

[1] The term *value* is used to refer to both integers and the *err* symbol.

tion or register satisfies an arithmetic or logical expression. For example, a detector can check if the value of register *$(5)* equals the sum of the values in register *$(3)* and memory location *(1000)* at a given program counter location. If the values do not match, an exception is thrown and the program is halted.

In our implementation, each detector is assigned a unique identifier, and the CHECK instructions encode the identifier of the detector they want to invoke in their operand fields. The detectors themselves are written outside the program, and the same detector can be invoked at multiple places within the program's code. We assume that the execution of a detector does not fail, i.e., the detectors themselves are error-free.

A detector is written in the following format:
det (ID, Register Name or Memory Location to Check, Comparison Operation, Arithmetic Expression)

The arguments of the detector are as follows:

(1) The first argument of the detector is its identifier.
(2) The second argument is the register or memory location checked by the detector.
(3) The third argument is the comparison operation, which can be any of ==, =/=, >, <, <= or >=.
(4) The final argument is the arithmetic expression that is used to check the detector's register or memory location and is expressed as:

*Expr :: = Expr + Expr | Expr – Expr | Expr * Expr | Expr / Expr | (c) | (Reg Name) | *(memory address)*

For example, the detector introduced above would be written as: *det(4, ($5), == , ($3) + *(1000)).*

5.4 Model Checking

Bounded model checking of programs [13] is performed using the *search* command in Maude [12]. The aim is to expose outcomes of the program caused by errors. The outcome is a user-defined function on the machine state described in Section 5.1 and must be specified as part of the *search* command. For example, the following search command obtains the set of all executions of the program that will print a value of *err* under all single errors in registers (one error per execution).

search regErrors(start(program, first, detectors)) =>!
(S:MachineState) such that (output(S) contains err) .

The *search* command systematically explores the search space in a breadth-first manner starting from the initial state and obtaining all final states that satisfy the user-defined predicate, which can be any formula in first-order logic. The programmer can query how specific final states were obtained or print out the search graph, which will contain the entire set of states that have been explored by the model checker. This can help the programmer understand how the injected error(s) lead to the outcome(s) printed by the search.

Termination: To ensure that the model checking terminates, the number of instructions that is allowed to be executed by the program must be bounded. This bound is referred to as the *timeout* and must be conservatively chosen to encompass the number of instructions executed by the program during all possible correct executions (in the absence of errors). After the specified number of instructions is exceeded, a "timed out" exception is thrown and the program is halted (we assume a watchdog mechanism is present).

6 Case Study

We have implemented SymPLFIED using Maude version 2.1. Our implementation consists of about 2000 lines of uncommented Maude code split into 35 modules. It has 54 rewrite rules and 384 equations.

This section reports our experience in using SymPLFIED on the *tcas* application [15], which is widely used as an advisory tool in air traffic control to ensure minimum vertical separation between two aircrafts (and hence avoid collisions). Other studies have extensively verified the safety of the *tcas* application from software defects (bugs) [16], but to the best of our knowledge, ours is the first study to verify *tcas* in the presence of hardware transient errors.

The *tcas* application consists of about 140 lines of C code, which is compiled to 913 lines of MIPS assembly code, which in turn is translated to 800 lines of SymPLFIED's assembly code (by our custom translator). *tcas* takes as input a set of 12 parameters indicating the positions of the two aircrafts and prints a single number as its output. The output can be one of the following values: 0, 1, or 2, where 0 indicates that the condition is unresolved, 1 indicates an upward advisory, and 2 indicates a downward advisory. Based on these advisories, the aircraft operator can choose to increase or decrease the aircraft's altitude.

In the later part of this section, we describe how we apply SymPLFIED on the *replace* program of the Siemens suite [18] in order to understand the effects of scaling to larger programs.

6.1 Experimental Setup

Our goal is to find whether a transient error in the register file during the execution of *tcas* can lead to the program's producing an incorrect output (in this case, an advisory). We chose an input for *tcas* in which the *upward advisory* (value of 1) would be produced under error-free execution. We directed SymPLFIED to search for runs in which the program did not throw an exception and produced a value other than 1 under the assumption of a single register error in each execution. The search space constitutes about (800 × 32) possible injections, since there are 32 registers in the machine, and each instruction in the pro-

gram is chosen as a breakpoint. To reduce the search space, at each breakpoint, only the register(s) used by the instruction was injected.

The injections were started on a cluster of 150 dual-processor AMD Opteron machines in order to ensure quick turn-around. The search command is split into multiple smaller searches, each of which sweeps a particular section of the program code looking for errors that satisfy the search conditions. The smaller searches can be performed independently by each node in the cluster and the results pooled together to find the overall set of errors. The maximum number of errors found by each search task was capped at 10, and a maximum time of 30 minutes was allotted for task completion (after which the task was killed).

To validate the results from SymPLFIED, we augmented the SimpleScalar simulator [17] with the capability to inject errors into the source and destination registers of all instructions, one at a time. For each register, we injected three extreme values in the integer range as well as three random values, so that a representative sample of the errors in each value can be considered by the injections.

6.2 SymPLFIED Results

Running Time: Of the 150 search tasks started on the cluster, 85 tasks completed within the allotted time of 30 minutes. The remaining 65 tasks did not complete in the allotted time. We report results only from the tasks that completed. Of the 85 tasks that completed, 70 tasks did not find any errors that satisfy the conditions in the search command (as either the error was benign or the program crashed due to the error). These 70 tasks completed within at most 1 minute. The time taken by the 15 completed tasks that found errors satisfying the search condition was less than 4 minutes, and the average time for task completion was 64 seconds. Even without considering the incomplete tasks, we were able to find the catastrophic outcome for *tcas*, shown below.

Outcomes: For the *tcas* application, we found only one case where an output of 1 is converted to an output of 2 by the fault injections. This can potentially be catastrophic, as it is hard to distinguish from the correct outcome of *tcas.* None of the other injections found any other such case. We also found cases where (1) *tcas* printed an output of 0 (unresolved) in place of 1, (2) the output was outside the range of the allowed values printed by *tcas,* or (3) the program crashed (numerous cases). We do not report these cases, as *tcas* is only an advisory tool, and the operator can ignore the advisory if he or she determines that the output produced by *tcas* is incorrect. We also found violations in which the value is computed correctly but printed incorrectly. We do not report these errors, as the output method may be different in the real system.

```
int alt_sep_test()
{
  enabled = High_Confidence && (Own_Tracked_Alt_Rate <=
OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
  tcas_equipped = Other_Capability == TCAS_TA;
  intent_not_known = Two_of_Three_Reports_Valid &&
(Other_RAC == NO_INTENT);
  alt_sep = UNRESOLVED;
  if (enabled && ((tcas_equipped && intent_not_known) ||
!tcas_equipped)) {
      need_upward_RA = Non_Crossing_Biased_Climb() &&
Own_Below_Threat();
      need_downward_RA =
Non_Crossing_Biased_Descend() && Own_Above_Threat();
      if (need_upward_RA && need_downward_RA)
        alt_sep = UNRESOLVED;
      else if (need_upward_RA)
        alt_sep = UPWARD_RA;
      else if (need_downward_RA)
        alt_sep = DOWNWARD_RA;
      else
        alt_sep = UNRESOLVED;
  }
    return alt_sep;
}
```

Figure 4: *tcas* code excerpt corresponding to error

Catastrophic Outcome Found by SymPLFIED: Figure 4 shows an excerpt from the *tcas* code. The code corresponds to the function *alt_sep_test*, which tests the minimum vertical separation between two aircrafts and returns an advisory. This function in turn calls the function *Non_Crossing_Biased_Climb()* and the function *Own_Above_Threat()* to decide if an upward advisory is needed for the aircraft. It then checks if a downward advisory is needed by calling the function *Non_Crossing_Biased_Descend()* and the function *Own_Below_Threat().* If neither or both advisories are needed, it returns the value *0* (unresolved). Otherwise, it returns the advisory computed.

The error under consideration occurs in the body of the called function *Non_Crossing_Biased_Climb()* and corrupts the value of register *$31*, which holds the function return address. Therefore, instead of control being transferred to the instruction following the call to the function *Non_Crossing_Biased_Climb()* in *alt_sep_test(),* the control gets transferred to the statement *alt_sep = DOWNWARD_RA* in the function. This causes the function to return the value *2* instead of the value *1,* which is printed by the program. Note that the above error occurs in the function call/return mechanism, which is added by the compiler. Hence, only a technique like SymPLFIED that reasons at the assembly language level can discover the error.

6.3 SimpleScalar Results

We performed over 6000 fault-injection runs on the *tcas* application using the modified SimpleScalar simulator to see if we could find any instances of the catastrophic outcome outlined in Section 6.1. In these experiments, we ensure that SimpleScalar is run for

the same time as SymPLFIED. Recall that SymPLFIED was run with 150 tasks, and each completed task took a maximum time of 4 minutes. This constitutes 10 hours in total, during which time we were able to perform 6000 automated fault-injection experiments using the SimpleScalar simulator. The results are summarized in column 2 of Table 2.

Table 2: SimpleScalar fault-injection results

Program Outcome	Percentage			
	# faults = 6253		*# faults = 41082*	
0 (Undecided)	1.86%	(117)	2.33%	(960)
1 (Upward)	53.7%	(3364)	56.33%	(23143)
2 (Downward)	0%	(0)	0%	(0)
Other	0.5%	(29)	1.0%	(404)
Crash	43.4%	(2718)	40.43%	(16208)
Hang	0.4%	(25)	0.8%	(327)

Even though we injected exhaustively into registers of all instructions in the program, SimpleScalar was unable to uncover even a single scenario with the catastrophic outcome of '2', whereas the symbolic error injection performed by SymPLFIED was able to uncover these scenarios with relative ease. This is because, in order to find an error scenario using random fault injections, not only must the error be injected at the right place in the program (for example, register *$31* in the *Non_Crossing_Biased_Climb* function), but also the right value must be chosen during the injection (for example, the address of the assignment statement must be chosen in Figure 4).

We also extended the SimpleScalar–based fault injection campaign to inject 41000 register faults in the *tcas* application. The injection campaign completed in 35 hours but was still unable to find such an error. The results of this extended set of injections is shown in column 3 of Table 2.

6.4 Application to Larger Programs

We analyzed the *replace* program using SymPLFIED. *replace* is the largest of the Siemens benchmarks [18] and is used extensively in software testing. The *replace* program matches a given string pattern in the input string and replaces it with another given string. The program translates to about 1550 lines of assembly code. Using the same experimental setup as described in Section 6.1, we ran SymPLFIED on the *replace* program to find all single register errors (one per execution) that lead to an incorrect outcome of the program. The overall search was decomposed into 312 search tasks. Of these, 202 completed execution within the allotted time of 30 minutes. In 148 of the completed search tasks, either the error was benign or the program crashed due to the error. Only 54 of the search tasks found error(s) leading to incorrect outcomes. The analysis completed in an average of 4 minutes where no erroneous solutions where found. Runs that found error(s) took 10 minutes on average.

7 Conclusions

This paper presented SymPLFIED, a modular, flexible framework for performing symbolic fault-injection and evaluating error detectors in programs. We have implemented the SymPLFIED framework for a MIPS-like processor using the Maude rewriting logic engine. We demonstrated the SymPLFIED framework on a widely deployed application, *tcas*, and found a non-trivial case of a hardware transient error that can lead to catastrophic consequences for the *tcas* system.

Acknowledgments: This research was funded in part by NSF grant CNS-05-51665 and CNS-04-6351. We would also like to thank the Gigascale System Research Consortium (GSRC) and the Motorola Center for Communications at UIUC for their support.

References

[1] Hiller, M., A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*, pp. 135-144, 2002.

[2] Arlat, J., et al. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.*, 16(2), pp. 166-182, February 1990.

[3] Cyrluk, D. Microprocessor verification in PVS: A methodology and simple example. Tech Report SRI-CSL-93-12, 1993.

[4] Boyer, R. S., and J. S. Moore. Program verification. *Journal of Automated Reasoning*, 1(1), pp. 17-23, 1985.

[5] Krautz, U., et al. Evaluating coverage of error detection logic for soft errors using formal methods. *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, 2006.

[6] Seshia, S. A., W. Li, and S. Mitra. Verification-guided soft error resilience. *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, 2007.

[7] Nicolescu, B., et al. On the use of model checking for the verification of a dynamic signature monitoring approach. *IEEE Trans. on Nuclear Science*, 52(5), pp. 1555-1561, October 2005.

[8] Perry, F., et al. Fault-tolerant Typed Assembly Language, *Int'l Conf. on Prog. Lang. Design and Implementation (PLDI)*, 2007.

[9] King, J. C. Symbolic execution and program testing. *Communications of ACM*, 19(7), pp. 385-394, July 1976.

[10] Bush, W., et al. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7), 2000.

[11] Larrson, D., and R. Hahnle. Symbolic fault injection. *International Verification Workshop (VERIFY)*, vol. 259, pp. 85-103, 2007.

[12] Clavel, M., S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. *Proc. First Int'l Workshop on Rewriting Logic and Its Applications*, 1996.

[13] Clarke, E., et al. Bounded model-checking using satisfiability solving. *Formal Methods in System Design*, July 2001.

[14] Pattabiraman, K., et al. SymPLFIED: Symbolic program-level fault Injection and error detection framework. Technical Report UILU-ENG-08-2205, University of Illinois at Urbana-Champaign, March 2008.

[15] Federal Aviation Administration. TCAS II Collision Avoidance System (CAS) System Requirements Spec, 1993.

[16] Lygeros, J., and N.A. Lynch. On the formal verification of the TCAS conflict resolution algorithms. *Proc. 36th IEEE Conf. on Decision and Control*, pp. 1829--1834, 1997.

[17] Burger, D., and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3), 1997.

[18] Hutchins, M., et al. Experiments on the effectiveness of dataflow- and control flow-based test adequacy criteria. *Int'l Conf. on Software Engineering (ICSE)*, pp. 191–200, 1994.