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, kalbar, rkiyer} @uiuc.edu

Abstract

This paper introduces SymPLFIED, a program-level framework which 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 symbolically 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.

1 Introduction

Error detection mechanisms are vital for building highly reliable systems. However, generic detection mechanisms such as exception handlers can take millions of processor cycles to detect errors in programs [3]. In the intervening time, the program can execute with the activated error and perform harmful actions such as writing incorrect state to the file-system. There has been significant work on efficiently placing and deriving error detectors for programs [1][2]. An important challenge is to enumerate the set of errors the mechanism fails to detect, either from 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 to evaluate the coverage of error detection mechanisms [4][5]. 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 paper shows how this can uncover possible "corner cases" which may be missed by conventional fault injection due to its inherent statistical nature. While there have been formal frameworks, each addresses a specific error detection mechanism (for example replication [12]), and cannot be easily extended to general detection mechanisms.

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 both 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 [17] 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.

Previous work [15] has analyzed the effect of hardware errors on programs expressed in a high-level language (e.g. Java). Errors are modeled as bit flips in single data variable(s) in the program. While this is an important step, there are several limitations, namely (1) low-level hardware errors can affect multiple program variables as well as impact the program's control-flow, (2) errors in special-purpose registers such as the *stack pointer* are difficult to model in the high-level language, (3) Errors in the language runtime system (and libraries) cannot be modeled as they may be written in a different language.

This paper considers programs represented at the assembly language level. The value of using assembly language is that any low-level hardware error that impacts 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.

It can be argued that in order to really analyze the impact of hardware errors, we need to model systems at even lower levels, *e.g* the register-transfer level (RTL). However, the consequent state space explosion when analyzing the entire program at such low levels can impact the practicality of the model. *An assembly language representation is a judicious tradeoff between the size of the model and the representativeness of hardware errors that can be considered in the model.*

In order to evaluate the framework, the effects of hardware transient errors are considered on a commercially deployed application, *tcas*. The framework identified errors that lead to a catastrophic outcome in the application, while a random fault injection experiment did not find any catastrophic scenario in a comparable amount of time. The framework is also demonstrated on a larger program, *replace*, to find instances of incorrect program outcomes due to hardware transient errors.

2 Related Work

Prior literature related to this work is classified into the following categories:

Error Detection: Many error detection mechanisms have been proposed in the literature, along with formal proofs of their correctness [1][10]. However, the verification methodology is usually tightly coupled with the mechanism under study. For example, [11] proposes and verifies a control-flow checking technique by constructing a hypothetical program augmented with the technique and model-checks 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 other error-detection mechanisms.

A recent paper [12] 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 [12] either accepts or rejects a program based on whether the program has been duplicated correctly, but 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 [13]. 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 [14]. However, Prefix assumes that the hardware does not experience errors during program execution.

Recently, a symbolic approach for injecting faults into programs was introduced in [15]. 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 [15] 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 for many important classes of programs.

Program verification techniques have been used to prove that a program's code satisfies a programmer-supplied specification [7]. The specification precisely outlines the expected result of the program given certain initial conditions. Typically, program verification techniques are geared towards finding software defects and assume that the hardware and the program environment are error-free. In other words, they prove that the program satisfies the specification *provided* the hardware platform on which the program is executed does not experience errors.

Further, program verification techniques operate on an abstract representation of the program (such as a state machine) extracted from the program code [23][24]. The abstractions are derived based on the specific property being checked and cannot be used for evaluating the program under arbitrary hardware errors as such errors may not manifest in the abstraction.

Formal techniques have also been extensively applied to **microprocessor verification** [6]. The techniques attempt to prove that the implementation of the processor conforms to an architectural specification usually in the form of a processor reference manual. Processor verification techniques focus on unmasking hardware

design defects, as opposed to transient errors due to electrical disturbances or radiation.

Soft-errors in Hardware: The techniques presented in [8] and [9] consider the effects of hardware transient errors (soft errors) on error-detection mechanisms 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 an error in an ASIC implementing the application.

Summary: The formal techniques considered in this section predominantly fall into the category of software-only techniques which do not consider hardware errors [7], or into the category of hardware-only techniques which do not consider the effects of errors on software [6]. Further, existing verification techniques are often coupled with the detection mechanism (e.g. duplication) being verified [11][12].

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. Moreover, the technique must be *automated* in order to ensure wide adoption, and should not require programmer intervention.

This paper attempts to answer the question: "*Is it possible to develop a framework to reason about the effects of arbitrary hardware errors on applications in an automated fashion, in order to understand where error detection mechanisms fail in detecting errors?*

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 the technique 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 flow of the SymPLFIED framework.

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

Assembly Language: 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 RISC processor architectures, it is currentl 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 independent of the Operating System (OS), and (2) Invocation of error detectors using special annotations, called *CHECK*, which allows detectors to be represented in line with the program.

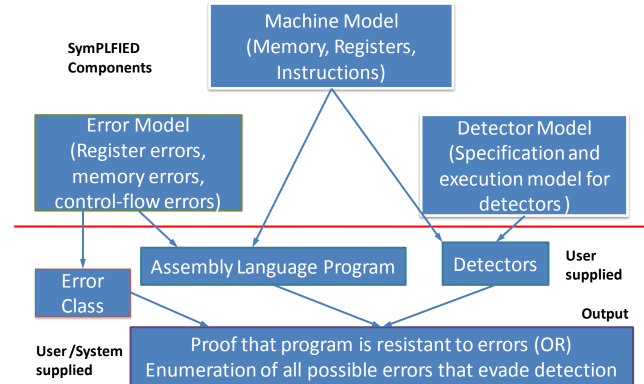

Figure 1: Conceptual design flow of SymPLFIED

Operation: The program behavior is abstracted using a generic assembly language described in Section 5. This is automatically translated into a formal mathematical model that can be represented in the Maude system [16]. 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 programmer intervention*.

Outputs: The framework uses the technique described in section 3.2 and outputs either of the following:

1. Proof that the program with the embedded detectors is resilient to the error class considered.
2. A comprehensive set of all errors belonging to the error class that evade detection and potentially lead to program failure (crash, hang or 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 pro-

gram 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, we can reason about the effects of errors (in the error model) on both programs, represented in the machine model and on detectors, represented in the detector model, in a unified fashion.

Correctness: In order 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 [16],

[2] The model must be an accurate representation of the system being modeled.

The first requirement can be satisfied by formally analyzing the specification using automated checking tools for each desirable property listed above. This is obtained almost for free by expressing the model using Maude as formal checking tools are available to check the conformance of the model to the properties [18].

However, the second requirement is much harder to ensure as it cannot be checked by formal tools and is usually left to the model creator. We have attempted to validate the model by rigorously analyzing the behavior of errors in the model and comparing them with the behavior of the real system (Section 6.3).

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 1). The symbol also introduces non-determinism in the program when used 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 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, would assign 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 that 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 is not always *accurate,* meaning that it may find false-positives.

Soundness is more important than accuracy from the point of view of designing detection mechanisms, as we can always augment the set of error detectors to conservatively protect against a few false-positives (due to the inaccuracies introduced).

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 the 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. The constraint solver also considerably limits state space explosion and quickly prunes infeasible paths [17]

3.3 Fault Model

The fault-model considered by SymPLFIED includes transient errors in memory/registers and computation.

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 pipelin*e and* how they affect the architectural state as shown in Table 1.

Errors in processor control-logic (such as in the register renaming unit) are not considered by the fault-model.

The reason it is possible to represent such a broad class of errors in the model is because the program is represented in assembly language, which makes the elements of its state explicit to the analysis framework.

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.

Table 1: Computation error categories and how they are modeled by SymPLFIED

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 Decode Errors

In spite of this limitation, model-checking techniques have been successfully scaled to large code-bases such as operating system kernels and web-servers [23][24]. These approaches consider only parts of the system that are relevant to the property being verified. The relevant code portions are typically extracted by static analysis. However, static analysis is not useful for dealing with runtime errors that may occur in hardware.

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 lends itself to 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 presented 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 by (*setgt $5 $3 $4)*. Since *$3* has the value *err* in it, it cannot be determined if 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 case. 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!*. All these outcomes are found by SymPLFIED.

```
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, or the program times out[2] and is stopped.

Complexity: Note that in order for a physical fault-injection approach to discover the same set of outcomes for the program as SymPLFIED, it would need to inject

[2] We assume that a watchdog mechanism is present in the program

into 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 though, 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.

False-positives: In the 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 conservatively assume that both the *true* and *false* cases are possible, as it does not distinguish between errors in different bit-positions of variables. Note that in practice, false-positives were not a major concern, as shown in section 6.2.

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 annotations as explained in Section 3.1. The error detectors together with their supporting instructions (*mov* instruction in line 8) are shown in bold.

```
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 inserted

The same error is injected as before in register *$3 (*the new line number is 11). As shown in Section 4.1, 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.

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 lesser 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 make an informed decision about handling the errors.

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

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 [16]. **Maude** is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications [16]. *The main advantage of Maude is that it allows a wide variety of formal analysis techniques to be applied on the same specification*.

Supporting Tools: In order to make programs for existing architectures compatible with SymPLFIED, we provide a facility to translate programs written directly in the target architecture's assembly language into SymPLFIED's assembly language. In theory, any architecture can be supported but for now we support only the MIPS instruction set. We also built a query generator to explore the behavior of the program under common hardware error categories. Note that while the SymPLFIED framework can support arbitrary error classes, pre-defined error categories allow programmers to verify the resilience of their programs without having to write complex specifications (or any specifications).

In this section, we describe the details of the machine, detector and error models and show how the resilience of programs to hardware errors can be verified through exhaustive search i.e. bounded model-checking.

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 hence requires rewrite rules.

Assumptions: The following assumptions are made by the machine model when executing a program.

- 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 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.
- 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 processor'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. The machine state is obtained by concatenating one or more of the machine elements in a single 'soup' of entities. For example, the soup, *PC(pc) regs(R) mem(M) input(In) output(out)* represents a machine state in which the (1) current program counter is denoted by *pc,* (2) register file is denoted by *R,* (3) memory is denoted by *M* and (4) input and output streams are *in* and *out* respectively.

Execute Sub-Model: We consider example instructions from each instruction class and illustrate the equations used to model them. These equations are defined in the *execute* sub-model and use primitives defined in other sub-models (e.g. the *fetch* primitive).

1. **Arithmetic Instruction:** Consider the execution of the *addi* instruction, which adds the value[3] *v* to the register given by *rs* and stores the results in register *rd.* In the equation given below, the <_,_> operator represents the machine state obtained by executing an instruction (given by the first argument) on a machine state (given by the second argument). *C* represents the code of the program and is written outside the state to enable faster rewriting by Maude (as it is assumed to be immutable). The {_,_} groups together the code and the machine-state.

```
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 can 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 accordingly (by the error model).

3. **Load/Store Instructions:** Consider the example of the *ldi rt, rs, a* which loads the value in the memory location at the address given by adding the offset *a* to the value in the register *rs.* However, the load address needs to be checked for validity before loading the value. This is done by the *isValid* primitive (defined in the *Memory Submodel).*

```
eq { C , < ldi rt rs a , PC(pc) regs(R) mem(M) S > =  if ( isVa-
lid(R[rd] + a, M)  )  then { C, < fetch(C, pc), C(next(pc)) mem(M)
```

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

```
regs(R[rt] <- M[a + R[rs]]) > } else { C, < throw “Illegal addr”,
PC(next(pc)) mem(M) regs(R) > } fi .
```

4. Input/Output Operations: Input and output operations are supported natively on the machine since the operating system is not modeled. An example is the print instruction whose equation is as follows:

```
eq { C, < print rs, PC(pc) regs(R) output(O) S > } = { C, < fetch(C,
    pc), PC(next(pc)) regs(R) output(O << R[rd]) S > } .
```

5. Special Instructions: These instructions are responsible for starting and stopping the program. e.g. *halt* and *throw* instructions to terminate the program. The halt instruction transforms the super-state prior to their execution into a machine state in order to facilitate the search for final solutions by the model-checker (section 5.4). Its equation is given by:

```
eq { C, < halt , PC(pc) S > } = PC(done) S .
```

5.2 Error Model

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

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 inject 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 described in Section 5.1 The choice of which register or memory location to inject into is made non-deterministically by the injection sub-model using rewrite rules.

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 also described by equations as shown below. In the equations that follow, *I* represents an integer.

```
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-zero” else err fi
```

In other words, any arithmetic operation involving the *err* value also evaluates to *err (*unless it is multiplied by 0). Note also how the divide-by-zero case is handled in 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 used in section 5.1 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. In the *true* case of the *isEqual* primitive, the location being compared can be updated with the value it is being compared to. However, the *false* case is not as simple, as it needs to “remember” that the location involved in the comparison is not equal to the value it is being compared with. The same issue arises in the case of non-equality comparisons, such as *isGreaterThan, isLesserThan, isNotGreaterThan* and *isNotLesserThan.*
The *constraint tracking and solving* sub-model remembers these constraints and determines if a set of constraints is satisfiable, and if not, truncates the state-space exploration for the case corresponding to the constraint. This helps avoid reporting false-positives.

Constraint Tracking and Solving Sub-Model: A new structure called the *ConstraintMap* is added to the machine state in Section 5.1. The *ConstraintMap* structure maps each register or memory location containing *err* to a set of constraints that are satisfied by the value in the location. An example of a set of constraints for a location is the following: *notGreaterThan(5) notEqualTo(2) greaterThan(0).* This indicates that the location can take any integer value between 0 and 5 excluding 0 and 2 but including 5. The constraints for a location are updated whenever a comparison is made based on the location if and only if it contains the value *err*. Constraints are also updated by arithmetic and logic operations in the program.
For a given location, it may not be possible to find a value that satisfies all its constraints simultaneously. Such constraints are deemed un-satisfiable and the model-checker can terminate the search when it comes to a state with an un-satisfiable set of constraints (such a state represents a false-positive). The constraint solver determines whether a set of constraints is un-satisfiable and eliminates redundancies in the constraint-set.

Memory- and Control Handling Sub-Model: Memory and Control errors are also handled non-deterministically using rewrite rules 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 location or register satisfies an arithmetic or logical expression. For example, a detector can check if the value of register *$(5)* equals the sum of the values in the register *$(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 free of errors.
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 in the following format:

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

Using the above notation, the detector introduced earlier would be written as:

*det(4, $(5), == , ($3) + *(1000)).*

The equations for the detector's execution are independent of the equations in the machine model, and hence are not affected by errors introduced in the machine other than those that are present in the registers or memory locations used in the detector's expression. Execution of a detector also updates the constraints for the location being checked in the *ConstraintMap* structure described in section 5.2.

5.4 Model-checking

The exhaustive *search* feature of Maude is used to model-check programs [16]. The aim of the search command is to expose interesting "outcomes" of the program caused by errors in a particular category. The "outcome" is a user-defined function on the machine state described in Section 5.1 and must be specified in the *search* command. For example, the following search command obtains the set of executions of the program that will print a value of *err* under all single errors in registers (one 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 checking. This can help the programmer understand how the injected error(s) lead to the outcome(s) printed by the search.
Termination: In the absence of errors, most programs can be modeled as finite-space systems provided (1) they terminate after a finite amount of time or (2) they perform repetitive actions without terminating but revisit states. However, errors can cause the state space to become infinitely large, as the program may loop infinitely due to the error, never revisiting earlier states. In practice, this is impossible, since the program data is physically represented as bits and there are only a finite number of bits available in a machine. However, the state space would be so large that it is practically impossible to explore in full.
In order 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 that the processor has a watchdog mechanism.

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 [21][22], which is widely used as an advisory tool in air traffic control for ensuring minimum vertical separation between two aircrafts and hence avoid collisions. The 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). In the later part of this section, we describe how we apply SymPLFIED on the *replace* program of the Siemens suite to understand the effects of scaling to larger programs.

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.

6.1 Experiment Setup

Our goal is to find whether a transient error in the register file during the execution of *tcas* can lead to the program 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 command is identical to the one shown in section 1.

This constitutes about (800 * 32) possible injections, since there are 32 registers in the machine, and each instruction in the program is chosen as a breakpoint. In order to reduce the search space, at each breakpoint, only the register(s) used by the instruction was injected. This ensures that the fault is activated in the program.

In order to ensure quick turn-around time for the injections, they were started on a cluster of 150 dual-processor AMD Opteron machines. 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 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).

In order to validate the results from SymPLFIED, we augmented the Simplescalar simulator [20] 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

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* and (3) numerous cases where the program crashed. 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 consider these cases as the output method may be different in the commercial implementation of *tcas*.

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 (as the timeout chosen was too large). 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 1 minute overall.

The time taken by the 15 completed tasks that found errors satisfying the search condition, (including the catastrophic outcome) is less than 4 minutes, and the average time for task completion is 64 seconds. Even without considering the incomplete tasks we were able to find the catastrophic outcome for *tcas*, shown below.

Initially, we were surprised by the unusually low number of catastrophic failures reported in *tcas*. However, closer inspection revealed that the code has been extremely well-engineered to prevent precisely these kinds of error from resulting in catastrophic failures. The *tcas* application (and system) has been extensively verified and checked for safety violations by multiple studies [21]. Nevertheless, the fact that SymPLFIED found this failure at all is testimony to its comprehensive evaluation capabilities. Further, this failure was not exposed by the injections performed using Simplescalar. In order to understand better the error that lead to *tcas* printing the incorrect value of 2, we show an excerpt from the *tcas* code in Figure 4.

Optimizations: In order to reduce the number of states explored by the model-checker, we inject errors only into the registers used in each instruction of the program. Further, we inject the error just before the instruction that uses the register, in order to ensure fault

activation. The effect of the injection is equivalent to injecting the register at an arbitrary code location so that the error is activated at the instruction.

```
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: Portion of *tcas* code corresponding to error

Catastrophic Outcome Reported by SymPLFIED:
The code shown in Figure 4 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 *Own_Above_Threat()* function 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 advisory is needed or if both advisories are needed, it returns the value *0* (unresolved). Otherwise, it returns the advisory computed in the function.

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. We have verified that the error exposed above corresponds to a real error and is not a false-positive by injecting these faults into the augmented Simplescalar simulator.

Note that the above error occurs in the stack, which is part of the runtime support added by the compiler. Hence, in order to discover this error, we need a technique like SymPFLIED that can reason at the assembly language (or lower) level.

6.3 SimpleScalar Results

We performed over 6000 fault-injection runs on the *tcas* application using the modified Simplescalar simulator to see if we can find the catastrophic outcome outlined above. We ensured that both SymPLFIED and Simple-scalar were run for the same time to find these outcomes. The SymPLFIED injections were run with 150 tasks, and each completed task took a maximum time of 4 minutes. This constitutes 10 hours in total. We were able to perform 6000 automated fault-injection experiments with Simplescalar in that time. The results are summarized in column 2 of Table 2.

Table 2: SimpleScalar fault-injection results

Program Outcome	Percentage			
	# faults = 6253		*# faults = 41082*	
0	1.86%	(117)	2.33%	(960)
1	53.7%	(3364)	56.33%	(23143)
2	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)

Table 2 shows that 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 the *alt_sep_test* function in Figure 4. Otherwise the program may crash due to the error or the error may be benign in the program.

We also extended the SimpleScalar based fault injection campaign to inject 41000 register faults to check if such an injection discovers errors causing the catatrophic outcome. The injection campaign completed in 35 hours but was still unable to find such an error. The results of this extended set of injections in shown in column 3 of Table 2.

6.4 Application to larger programs

In order to evaluate the effectiveness of the formal analysis as we scale to larger applications, we analyzed the *replace* program using SymPLFIED. *replace* is the largest of the Siemens benchmarks [19], 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 code translates to about

1550 lines of assembly code spanning 22 functions. The key functions are listed in Table 3.

Table 3: Important functions in *replace*

`makepat`	Constructs pattern to be matched from input reg exp
`getccl`	Called by `makepat` when scanning a '[' character
`dodash`	Called by `getccl` for any character ranges in pattern
`amatch`	Returns the position where pattern matched
`locate`	Called by `amatch` to find whether the pattern appears at a string index

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.

Results: 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, while 54 of the search tasks found error(s) leading to incorrect outcome. We consider the execution trace of an example error.

Example Scenario: An input parameter to the `dodash` function that holds the delimiter (']') for a character range was injected. An erroneous pattern is constructed, which leads to a failure in the pattern match. As a result, the program returns the original string without the substitution. The analysis completed in an average of 4 minutes where no erroneous solutions where found. For the injection runs that found an erroneous outcome the analysis took an average of 10 minutes.

7 Conclusion

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 demonstrate the SymPLFIED framework on a widely-deployed application *tcas,* and use it to find a non-trivial case of a hardware transient error that can lead to catastrophic consequences for the *tcas* system. We also demonstrate the framework on the *replace* program, which is the largest among the Siemens programs.

Future work will include (1) Extending the SymPLFIED framework to other architectures than MIPS, (2) Modeling permanent errors in hardware in addition to transient errors, (3) Augmenting the design of the constraint solver to reduce false-positives and (4) Investigating intelligent state-space pruning technique to scale SymPLFIED to large programs.

Acknowedgements: We thank Long Wang and other members of the DEPEND group for their insightful comments about this paper. The first author would like to thank Shuo Chen for exposing him to Maude, and Grigore Rosu and Jose Meseguer for teaching him the finer points of Maude.

References

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

[2] Pattabiraman, K., Kalbarczyk, Z., and Iyer, R. K. Automated Derivation of Application-aware Error Detectors using Static Analysis. In *Proc. of the 13th Intl. on-Line Testing Symposium,* 2007.

[3] W. Gu, Z. Kalbarczyk, R.K. Iyer, Z. Yang, Characterization of Linux Kernel Behavior under Errors, *Proc. International Conference on Dependable Systems and Networks (DSN'03)*, pp. 459-468, 2003.

[4] Arlat, J., et al. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Trans. Softw. Eng.* 16, 2 166-182, Feb 1990.

[5] H. Madeira, J. Carreira, J.G. Silva. Injection of Faults in Complex Computers. *IEEE Workshop on Evaluation Techniques for Dependable Systems.* San Antonio. Texas. October 1995

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

[7] R. S. Boyer and J S. Moore. "Program Verification". *Journal of Automated Reasoning* 1, 1 (1985), 17-23.

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

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

[10] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *Intl. Conference on Distributed Computing Systems,* pages 436--443, May 1998.

[11] Nicolescu, B. Gorse, N. Savaria, Y. Aboulhamid, E.M. Velazco, R., On the use of model checking for the verification of a dynamic signature monitoring approach, IEEE Trans. on Nuclear Science, Vol. 52, 5(2), pp. 1555-1561, Oct 2005.

[12] Perry F., et al., Fault-tolerant Typed Assembly Language, *Proc. of Conf. on Prog. Lang. Design and Implementation (PLDI)*, 2007.

[13] King, J. C. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (Jul. 1976), pp. 385-394.

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

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

[16] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In, *Proc. First Intl. Workshop on Rewriting Logic and its Applications,* 1996.

[17] E. Clarke, A. Biere, R. Raimi, Y. Zhu. Bounded Model-Checking using satisfiability solving. *In Formal Methods in System Design*, July 2001.

[18] M. Clavel et al., The Maude Formal Tool Environment, *Springer Verlag LNCS*, Vol 4624, pp. 173-178, Aug 2007.

[19] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In Proc. of the *Intl. Conf. on Software Engineering (ICSE)*, pp 191–200, 1994

[20] Burger, D. and Austin, T. M. 1997. The SimpleScalar tool set, version 2.0. *Comput. Archit. News* 25, 3, 1997.

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

[22] Federal Aviation Administration, TCAS II Collision Avoidance System (CAS) System *Requirements Specification,* March 1993

[23] Ball, T. and Rajamani, S. K. The SLAM Toolkit. In Proc. Intl. Conf. on Computer Aided Verification (2001).

[24] H. Chen, D. Dean, and D. Wagner. Model-checking one million lines of C code. In Network and Distributed System Security Symposium, pages 171- 185, February 2004.