A Formal Verification Approach to Safeguard Controller Variables from Single Event Upset

Ganesha[1][0000−0001−9461−9231] and Sujit Kumar Chakrabarti[2][0000−0001−8422−2900]

[1] Samsung R&D Institute India-Bangalore
[2] `ganesha@iiitb.ac.in`

[3] International Institute of Information Technology, `{ganesha, sujitkc}@iiitb.ac.in`

Abstract. We present a method based on program analysis and formal verification to identify *conditionally relevant variables* (CRVs) – variables which could lead to violation of safety properties in control software when affected by *single event upsets* (SEUs). Traditional static analysis can distinguish between relevant and irrelevant variables. However, it would fail to take into account the conditions specific to the control software in question. This can lead to false positives. Our algorithm employs formal verification to avoid false positives. We have conducted experiments that demonstrate that CRVs indeed are fewer in number than what traditional static analysis can detect and that our algorithm is able to identify this fact. The information provided by our algorithm could prove helpful to a compiler while it does register allocation during the compilation of the control software. In turn, this could cause significant reduction in the cost of controller chips.

Keywords: Program Analysis, CRV (Conditional Relevance of a Variable), SEU (Single Event Upset), Hardening, Model Checking, Formal Verification, Program Slicing

1 Introduction

1.1 Single Event Upsets

Control systems are typically implemented as control algorithms running on a microcontroller or *system-on-chip* (SoC) devices. Often, such embedded controllers function in harsh environmental conditions. In such cases, the computing components get exposed to a rare random phenomenon called *single event upset* (SEU) [1]. When an SEU happens, a bit in the chip would flip its value. This could lead the ongoing computation to fall off track leading to errors – often with catastrophic consequences in safety critical applications, e.g. automotive, industrial automation, aerosphere, oil and gas, mining, healthcare etc.

1.2 Hardening

A common technique used in the industry to safeguard electronic components from SEUs is called *hardening* [2]. The hardened part of the silicon can withstand the causes of SEUs, and bit-flips are avoided. Hardening causes the cost of the chip to go up. Hence, it would be better to harden only a portion of the chip that balances the risk of SEUs with the rise in manufacturing costs of the chip. However, this would require that code running on the chip to be generated with this fact taken into account. During runtime, the parts of the code that must be protected from SEUs must reside in the hardened part of the chip while the other 'non-critical' components of computation can reside in the non-hardened part of the chip. In this paper, we present a method to statically identify variables which are 'critical' in the sense that their getting affected by SEUs would likely throw the computation off-track leading to unacceptable errors.

1.3 Our Contribution

In this paper, we present an approach for dividing the variables of the control algorithm into two categories: *conditionally relevant variables* (CRVs) – those which should be placed in the hardened part of the silicon, and the other – called *conditionally irrelevant* or non-CRVs – that can be left unguarded by being placed in the unhardened part. We note that discovering the precise set of CRVs is undecidable. Static analysis techniques, e.g., slicing, can be used to detect CRVs, however they give a sound but overapproximate set of CRVs. Our algorithm uses formal verification (model checking) that helps us tighten this bound without compromising soundness.

This paper makes the following contributions:

1. We introduce and formally define the idea of *conditional relevance* of a variable.
2. We use a combination of static analysis (program slicing) and formal verification (software model checking) to identify the conditional relevance/irrelevance of a variable.
3. We present experimental evaluation of our approach on a number of example programs and demonstrate the fact that:
   (a) There indeed are present conditionally irrelevant variables in control softwares.
   (b) Using formal verification over and above traditional static analysis (i.e. static program slicing), helps discover additional conditionally irrelevant variables, avoiding false positives and over-conservative decisions.

We believe that the method we present in this paper can help bring down the manufacturing cost of computing chips, especially in control software domain, by providing useful inputs to the compiler. Figure 1 illustrates the selective hardening process, where a program undergoes *analysis* (i.e. using the algorithm presented in this paper) to identify the critical parts which cannot be allowed to suffer SEUs. The information about these parts are passed on to the compiler which generates code to ensure that variables reside in the hardened and non-hardened parts of the chip during runtime based on their criticality.

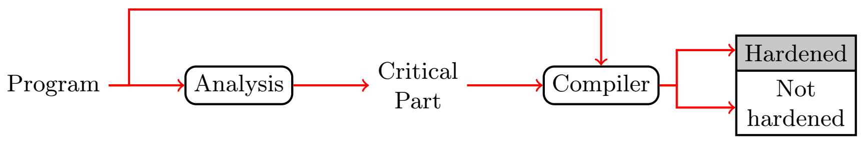

Fig. 1. Hardening

2 Motivating Example

```
int f(int x, int y) {
  int output = 4;
  bool alarm = false;
  int count = 0;
  while(count < 7) {
    if(x > 10) {
      if(y == 1) {
        output = 2;
      }
      else {
        output = 1;
      }
    }
    else {
      output = output + 1;
      alarm = true;
    }
    count++;
  }
  printf("%b", alarm);
  return output;
}
```

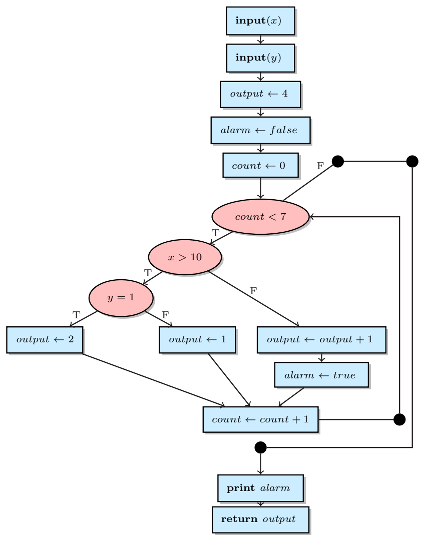

Fig. 2. Motivating Example

In Figure 2, we show a fragment of code on the left side and its control flow graph on the right side. The program computes and returns the value of the output variable *output*. Suppose that there is correctness condition imposed on the system that says that *output* should be less than or equal to 10 before it is returned. We can see that it is possible to violate this condition in the given code. The program iterates through the outer while loop exactly 7 times. If the branch condition ($x > 10$) is false, *output* would be incremented by 1 in each iteration. In 7 iteration, its value would be 11, which would violate the safety property.

But, an interesting aspect of this situation is that the above computation never uses the value of y. This is not to say that *output* is not dependent on y; in fact, it *is* dependent on y. A traditional static slicing would reveal this to us. But this technique would fail to discover that *output* doesn't depend on y's value when it violates the safety property. This is to say that y is not relevant as far as the safety property $output \leq 10$ is concerned. We use formal verification to identify such cases.

3 Terms and Concepts

3.1 Dependencies and Static Program Slicing

In a program, when one part, say A, affects the computation happening at some other part, say B, there is said to be a dependency between A and B, or B is dependent on A. There are two types of dependencies: *data dependency* and *control dependency*. When the computation at A is used as input for B, B is said to be *data dependent* on A. When the execution of B depends on the computation at A, B is said to be *control dependent* on A.

A *program slice* (or *slice* for short) in a program is the part of the program related in some causal way to a specific program location. For example, a backward slice gives those parts of the program which, during some run, influence the value of a variable at a program point of interest. This pair of variable (v) and program point (l) is known as the *slicing criterion*. Program slicing includes a range of techniques in program analysis to discover slices, which can be static, dynamic, backward, or forward. Slicing has been successfully and widely used to solve many software engineering problems.

3.2 Observations

Static slicing yields sound results, i.e. it will not miss identifying any CRV. However, it will often yield imprecise results. For example, in the Figure 2, it will not be able to detect the fact that y is not a CRV for $output > 10$ at the return statement. In this work, we use program slicing and formal verification to identify the conditionally relevant variables. As a result, the compiler can be guided to place conditionally relevant variables in the hardened registers during register allocation. These bounds will often be tighter than the ones obtainable through traditional static analysis. This will make it possible to harden a smaller part of the chip, leading to cost saving without loss of safety guarantee.

3.3 The safety property Φ

A safety property defines the safety of the computation and is often a weaker condition than strict correctness. This is because, in close-loop feedback control systems, many glitches (e.g. those arising out of SEUs) do not persist beyond a control cycle and are often self-correcting due to the dynamic properties of the controlled plant. In other words, the plant often acts like a low-pass filter that removes high frequency transients before they can adversely affect the system function.

As an example, suppose that in a temperature control system, if the value of the variable o corresponding to the output heating rate gets affected by SEU, the plant temperature θ is not going to change very fast due to the high time constants associated with heating. As a result the next value of the same variable o, computed based on the error value $(\theta_{ref} - \theta)$, will not be significantly influenced by the fact that it had got disturbed due to an SEU in the last control cycle. The system is capable of 'forgetting' one-time upsets like this in many cases. Any exception (either internal or environmental) causing value changes in o would cause *unacceptable* changes in the controlled plant only if it persists for some time. The safety property formulates the criterion that defines an acceptable or safe behaviour from the point-of-view of the controlled plant.

More formally, the safety property is a function that defines if the output computed at the *output point*, i.e. end of each control cycle, is 'safe'. This is derived from knowledge about the physical properties of the plant being controlled. For now, this process is based on domain knowledge.

For example, it may be stipulated that the computed control output o must not be outside a certain range, say (r_{min}, r_{max}) for more than 5 consecutive times. Suppose O is a buffer which is used to store the last 5 values of the output variable, with O_4 being the last computed value and O_0 being the fifth most recent value of o. Then,

$$\Phi(O) = \neg(\bigwedge_{i=0}^{4} (O_i < r_{min} \lor O_i > r_{max}))$$

3.4 Conditionally Relevant Variable (CRV)

We now present the idea of conditionally relevant variables by first giving an informal explanation and then presenting a formal definition.

Relevant variables are those on which the output of the computation is dependent. Static analysis, e.g. program slicing, can identify relevant variables. On the other hand, a *conditionally* relevant variable (CRV) is one which affects the satisfaction of a given safety property Φ. Conversely, a *conditionally irrelevant variables* (non-CRV) is one whose value does not influence the satisfaction of the given safety property. Therefore, a variable may be relevant, but may be conditionally irrelevant as per the given safety property.

More specifically, a conditionally relevant variable is one whose value – if changed randomly during program execution – would lead to a change in the value of the safety property in at least one possible execution.

Now, we present a more formal definition of conditional relevance.

Suppose that V is the set of all variables in the controller program. Let $I(\subseteq V) = \{x_1, x_2, .., x_n\}$ be the set of all input variables of the program, i.e., their values are set through interactions with the environment, e.g. controlled plant, human operator/user, sensors or other subsystems. Let's say that, for a given run of the program, the input values received by an input variable x_i are $\mathcal{I}(x_i) = \{x_{i,1}, x_{i,2}, ...\}$. We define an input vector as a sequence of all values assigned to all input variables: $\mathcal{I} = \bigcup_{x_i \in I} \mathcal{I}(x_i)$. We denote the set of all possible input vectors by $\mathbf{I}$.

A trace is the sequence of program locations (possibly with multiple occurances due to loops) visited during an execution of the program. In absence of any upsets, each input vector would map to a unique execution trace, given by $\pi(\mathcal{I}) = \{l_1, l_2, ...\}$ during the execution corresponding to $\mathcal{I}$. The output location is the program point p that immediately precedes the command that causes the result variable to be output by the controller in each control cycle. An output point set $P(\mathcal{I}) \subseteq \pi(\mathcal{I})$ for an input vector $\mathcal{I}$ is the set of occurances of the output location p in $\pi(\mathcal{I})$.

Let $l \in \pi(\mathcal{I})$ be the location where a variable $x \in V$ suffers an SEU. As a result, the trace that evolves is $\pi'(\mathcal{I})$. We note that this may cause $\mathcal{I}$ to no more be completely input to the program due to the modified trace possibly diverging away from the original. However, the prefix of the modified input vector $\mathcal{I}'$ upto l would be identical to that of $\mathcal{I}$, and that is what we are interested in. Correspondingly, the output point set thus visited is $P'(\mathcal{I}) \subseteq \pi'(\mathcal{I})$.

Formally, the conditional relevance of a variable x is defined as:

$$\begin{aligned} CRV(x) = \exists\, \mathcal{I} \in \mathbf{I} \text{ such that } & \\ & (\forall p \in P(\mathcal{I}) \mid \Phi_p = true \wedge \exists p' \in P'(\mathcal{I}) \mid \Phi'_p = false) \\ & \qquad\qquad \vee \\ & (\forall p' \in P'(\mathcal{I}) \mid \Phi'_p = true \wedge \exists p \in P(\mathcal{I}) \mid \Phi_p = false) \end{aligned} \tag{1}$$

In other words,

1. Conditional relevance requires the safety property to be violated for at least one input vector when x is affected by an SEU ($\Phi'_p = false$) in the condition that the safety property is never violated for that input vector in absence of an SEU affecting x ($\Phi_p = true$).
2. Also, conditional relevance requires the safety property to hold true in presence of SEU on x while there is violation in its absence.

When an SEU affects x, the first condition above leads to an occurance of fault, while the second case could cause a fault getting masked due an SEU. Both these situations are undesirable. Please note that the second case is likely to be rarer, but possible.

All variables $x \in V$ for which $CRV(x) = true$ are conditionally relevant variables. All other variables are conditionally irrelevant variables.

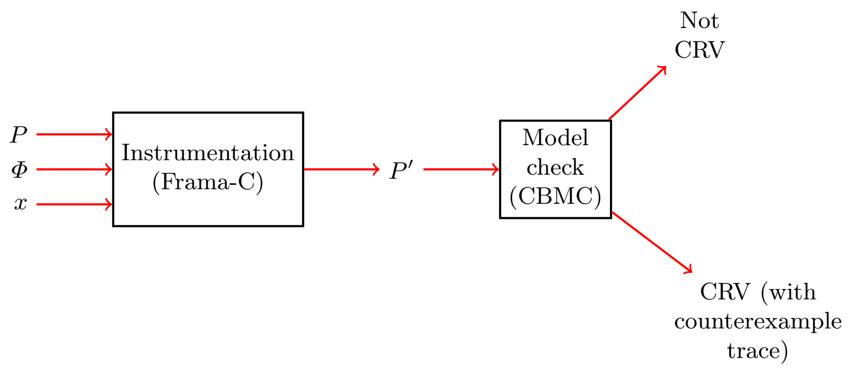

Fig. 3. Overall approach

4 Our Approach

4.1 Overall Architecture

Figure 3 shows our overall approach. P is the control program that is being analysed, Φ is the safety property and x is the *variable under investigation*, i.e. we are investigating if x is a CRV or not. The first step is to instrument P and generate the instrumented version P'. This is given as an input to a model checker. The model checker either establishes that x is not a CRV, or establishes that it is a CRV, along with a counterexample trace through P' that shows how an SEU affecting x may lead to the violation of the safety property.

4.2 Instrumentation

Our overall approach involves instrumenting the control procedure P in preparation for verification. We call the resultant instrumented procedure P'. Consider the structure of P as shown in Algorithm 1. I is the initialisation part of the program. B is the part of the main loop which computes the output variable o. The sequence S is created by taking B and sequencing with it O`.append(`o`)` and ϕ `:=` $\Phi(O)$. As P' is created by merging the original and modified version of P, we ensure to replace all variables in P with another variable, in both I and S, denoted by $I' \leftarrow I[V'/V]$ and $S' \leftarrow S[V'/V]$ respectively. Here, V is set of variables in P and V' is the set of new variables. O is a buffer that is used to maintain the last n values of the output variable o. O`.append(`o`)` results in o being added as the last element of O. If O already had n elements, its first element is dropped in order to make space for the latest value of o. A variable under investigation, say x, may suffer from an SEU anytime during the execution of the program. Hence, we insert a `simulate_seu` command (see section 5) before every use of x.

Algorithm 1 Instrumentation algorithm

function INSTRUMENT(P)

▷ P is a control algorithm with the following structure:

$$P = \begin{tikzcd} \boxed{I} \arrow[r] & \texttt{true} \arrow[r] & \boxed{B} \arrow[r] & \boxed{\texttt{return } o} \arrow[ll, bend left] \end{tikzcd}$$

▷ where B is the body of an arbitrary control computation.

$$S \leftarrow \begin{tikzcd} \boxed{B} \arrow[r] & \boxed{O\texttt{.append}(o)} \arrow[r] & \boxed{\phi \texttt{ := } \Phi(O)} \end{tikzcd}$$

$I' \leftarrow I[V'/V]$
$S' \leftarrow S[V'/V]$

return

$$\begin{tikzcd} \boxed{I; I'} \arrow[r] & \texttt{true} \arrow[r] & \boxed{S} \arrow[r] & \boxed{S'} \arrow[r] & \boxed{\texttt{assert}(\neg(\phi \oplus \phi'))} \arrow[lll, bend left] \end{tikzcd}$$

4.3 Simulating the SEU

To simulate SEU, we inject bit flip on the variable under investigation before each of its use. This is done through code instrumentation. We have developed a tool that is built upon C Intermediate Language (CIL) [12] that does the instrumentation automatically. This inserts calls to the helper function `simulate_seu` into the instrumented program. A simplified version of the logic of inserting bit-flip is shown in Figure 5. SEU can happen at any place of the program. We also ensure that the SEU can happen up to only once (using `flag` variable). SEU simulation method is explained in Figure 4.

Figure 5 shows the code for injecting a Single Event Upset (SEU) into a target variable. A single-bit flip is introduced at a random bit position using XOR logic. Instrumentation preserves original semantics of the program apart from the calls to the function `simulate_seu_main`. We call `simulate_seu_main(&x)` before every use of `x`, where `x` is the variable under investigation. The function internally uses `nondet_int()` from CBMC to mimic random bit flip.

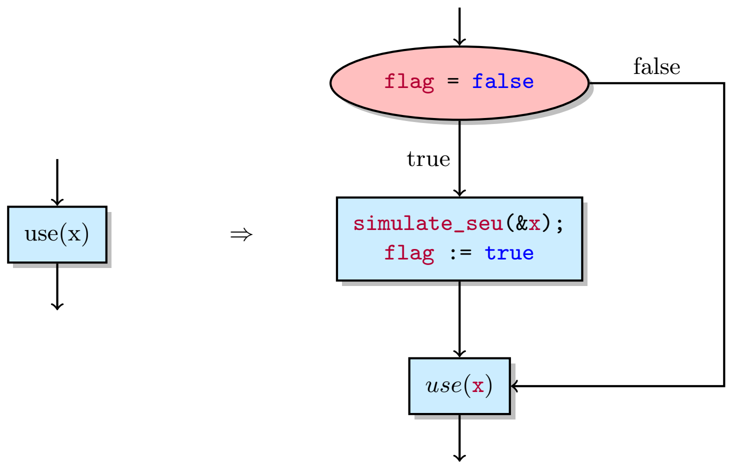

Fig. 4. SEU Simulation Method

5 Experimental Results

5.1 Experimental Setup

The main goal of our experiment is to validate and demonstrate the effectiveness of our proposed algorithm. We use a set of controller algorithms written in C language as case studies, as shown in Table 1. We have used Frama-C version 30.0 [13] for program slicing and CBMC version 5.12 for model checking.

```
// Function to generate a nondeterministic integer within the
// range [1, 32]
int nondet_int_range_1_32() {
    int value = nondet_int() % 32 + 1;
    return value;
}

int simulate_seu(int value, int bit_pos) {
    int mask = 1 << bit_pos;
    return (value ^ mask); // XOR operation for bit flip
}

// Ensures that an SEU is introduced only once for the
// variable under investigation
void simulate_seu_main(int *invest_var) {
    static int count = 0;
    if(count == 0) {
        int bit_pos = nondet_int_range_1_32();
        *invest_var = simulate_seu(*invest_var, bit_pos);
        count++;
    }
}
```

Fig. 5. Code to simulate SEU

The slicing criterion for each case study was selected based on a safety property defined in terms of the controller output variable at output point (i.e. the program location just before when output value is written to actuator/plant) of the controller procedure.

We first slice the input program to remove everything except the relevant program statements w.r.t. to the safety property. The sliced input code is fed into an instrumentation step (see Section 4.2). Here, we apply our SEU simulation method (Section 4.3) to the variable under investigation and create the instrumented version of the program. The instrumented code is then fed to a model checker (CBMC), along with the safety property. CBMC applies bounded model checking and determines whether the safety property is violated or not. If it is violated, CBMC provides a counterexample trace.

5.2 Case Studies and Evaluation

To evaluate the effectiveness of our algorithm we have tested it on one controller algorithms (*Motivating Example*) which mimics the real world closed-loop systems. Additionally, we have verified our algorithm on two programs (*Temperature Control* and *Fan Speed Control*) not having a structure similar to a typical control algorithm (i.e. with an outer while loop and control output being computed in the loop body) to demonstrate its general applicability. We have compared our algorithm with static slicing in terms its precision in detecting CRVs. We consider below points during our evaluation.

1. Total number of variables examined
2. CRVs identified in both the methods
3. Non-CRVs retained by static slicing but correctly eliminated by our approach
4. The safety property used for each program

Program Name	**LoC**	**T**	**S**	**M**	$\boldsymbol{\eta}$ (%)	$\boldsymbol{\Phi}$
Motivating Example	30	5	4	1	25%	$output \leq 10$
Temperature Control	45	6	4	2	50%	$temperature \leq 30$
Fan Speed Control	45	5	4	1	25%	$fan\_speed \leq 100$

Table 1. Comparison of CRV Detection: Static Slicing vs Our Approach

As shown in Table 1, some of the variables identified as relevant variables (RVs) by static slicing are found to be conditionally irrelevant (flagging them as non-CRVs) by our algorithm. For instance, in each of the *Motivating Example* and *Fan Speed Control* programs, our approach eliminates one such variable that static slicing retains. In the *Temperature Control* case, two additional non-CRVs were correctly discarded by our method. These results support our claim that the proposed approach improves over static slicing by reducing false positives and more precisely identifying the conditionally relevant variables.

We define the following parameters for the results shown in Table 1. T represents the total number of input variables in the program. S denotes the number of relevant variables retained by static slicing (i.e., RVs). M represents the number of non-CRVs detected and eliminated by our approach. η denotes the efficiency of our method, and Φ represents the safety property being verified for each benchmark. LoC indicates the number of lines of code in the program.

We calculate the efficiency η, as $\eta = \frac{M}{S} \times 100$, where S is the number of relevant variables retained by static slicing and M is the number of non-CRVs correctly eliminated by our approach. A higher value of η indicates a greater number of false positives successfully removed by our method, demonstrating its advantage over slicing. For example, if static slicing retains $S = 4$ variables and our algorithm eliminates $M = 1$ non-CRV, the efficiency is calculated as $\eta = \frac{1}{4} \times 100 = 25\%$.

6 Related Work

6.1 Program Slicing and Its Limitations

Program slicing is well known method of program analysis and debugging. Traditional program slicing, such as those introduced by Weiser [3] and surveyed by Tip [4], focuses on semantic dependencies to slice the program. However these approaches often lead to over approximation, i.e. they include the variables that semantically do not impact the program behaviour. Recent studies in program slicing explored some enhancements to existing slicing techniques. For example, [5] proposes predictive slicing using machine learning to anticipate runtime behaviour and [6] introduces field-sensitive slicing to improve precision. Despite these enhancements to program slicing, the actual challenge, i.e. of identifying actual set of variables that are really critical for program correctness, especially under controller safety propertys, still remains the same.

6.2 Software Model Checking and Verification Tools

Today, state-of-the-art software model checking is significantly evolved. Tools such as CBMC [7] and ESBMC [8] that can formally verify C and C++ programs. These tools are used to identify bugs and to verify the program correctness. Recently, CBMC added a symbolic shadow memory as an additional mechanism to verify memory safety [9]. Our work builds on top of software model checking to detect CRVs in controller algorithms.

6.3 SEU Mitigation Strategies

Since SEUs occur in control chips, they impact system reliability especially in the harsh environmental condition. Conventional mitigation approaches are concentrated on hardware level solutions such as error correcting codes and redundancy. [10] proposes a self-adaptive SEU mitigation scheme based on ECC and refreshing technique for embedded systems. [11] proposes a software-only fault tolerance technique, SWIFT, to detect and recover from transient faults such as SEUs and Single Event Transients (SETs) in microprocessors. Our work introduces a novel approach by leveraging static analysis and formal methods to identify CRVs.

7 Conclusion

In this paper, we have introduced the concept of conditionally relevant variables (CRVs) – variables whose values must not be affected by random changes at runtime to ensure safe execution of the control program. This is represented by the satisfaction of a safety property. Our algorithm determines the conditional relevance of variables in a control software using static analysis and formal verification. We have demonstrated that the set of CRVs is often a proper subset of all relevant variables. This fact could be used to provide more fine-grained control to the compiler so as to ensure that only CRVs reside in the hardened portions of the silicon. This would make the idea of partial hardening (i.e. hardening only portions of the chip) practical without compromising safety or performance. This, in turn, would help bring down chip manufacturing costs.

7.1 Future Work

Currently, we have restricted our attention to only variables in terms of their conditional relevance. The idea can be extended to the verification of conditional relevance of program locations too. Further, in its current form, our algorithm can analyse single-procedure control programs. In future we will extend our research to handle multi-procedure programs using inter-procedural analysis. Also, we plan to explore other model checking tools beyond CBMC to ensure soundness over the complete state space. Above enhancements will enable us to apply our algorithm to industrial-strength case studies. Finally, we aim to automate the formulation of safety properties based on system specification.

References

1. Jilani, A., Sharma, D., Naaz, R.: Single Event Upset. In: National Conference on Industry 4.0 (NCI 4.0) (2020). https://www.researchgate.net/publication/351515765_Single_Event_Upset, last accessed 2025/04/20
2. Science Alert: Overview of Radiation Hardening Techniques for IC Design. https://scialert.net/fulltext/?doi=itj.2010.1068.1080, last accessed 2025/04/20

3. Weiser, M.: Program Slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449 (1981)
4. Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages (1995)
5. Yadavally, A., et al.: Predictive Program Slicing via Execution Knowledge-Guided Learning. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) (2024)
6. Galindo, C., et al.: Field-Sensitive Program Slicing. Journal of Systems and Software (2024)
7. Kroening, D., Tautschnig, M.: CBMC–C Bounded Model Checker. In: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 389–391 (2014)
8. Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: Model Checking C++ Programs. Software Testing, Verification & Reliability **32**(3), e1793 (2022)
9. Fischer, B., et al.: CBMC-SSM: Bounded Model Checking of C Programs with Symbolic Shadow Memory. In: Proceedings of the International Conference on Automated Software Engineering (2023)
10. Lu, Y., et al.: A Self-Adaptive SEU Mitigation Scheme for Embedded Systems in Extreme Radiation Environments. IEEE Systems Journal **16**(1) (2022)
11. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software Implemented Fault Tolerance. In: Proceedings of the 2005 International Symposium on Code Generation and Optimization (CGO). `https://doi.org/10.1109/CGO.2005.34`
12. CIL: Infrastructure for C Program Analysis and Transformation, Version 1.7.3. `https://cil-project.github.io/cil/doc/html/cil/`
13. Frama-C: Slicing Plugin. `https://frama-c.com/fc-plugins/slicing.html`