2022 The 22st International Conference on Control, Automation and Systems (ICCAS 2022)
BEXCO, Busan, Korea, Nov. 27∼Dec. 01, 2022

Symbolic Fault Injection for Plan-based Robotics

Tim Meywerk[1*], Vladimir Herdt[1,2] and Rolf Drechsler[1,2]

[1]Group of Computer Architecture, University of Bremen, Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{tmeywerk, vherdt, drechsler}@uni-bremen.de
* Corresponding author

Abstract: Autonomous robots are being used increasingly in safety-critical environments. Due to their dynamic nature and uncertainty, failures of low-level actions are common. In plan-based robotics, these failures are handled inside the higher-level plans, using a multitude of failure handling strategies. With the increasing complexity of robotic plans, failures may be accidentally left unhandled. This will usually stop the plan entirely. To avoid such a situation, the failure handling should ideally be complete, i. e. there should be no failure that can reach the top-level of the execution.
In this paper, we propose to use formal methods, in particular symbolic fault injection to tackle the problem of finding unhandled failures or proving that no such failure exists. We implement symbolic fault injection for the CRAM Planning Language (CPL). We base our work on the worst-case assumption that any low-level action may fail at any time with any of its possible types of failure. Our work builds upon an existing symbolic execution engine for CPL and extends it to reason about CPL's failure handling mechanism. We also present a way to implement the worst-case assumption directly into CPL. Our experimental evaluation suggests that symbolic fault injection is a suitable and scalable method to find unhandled failures in robotic plans.

Keywords: Formal Verification, Autonomous Robots, Planning, Fault Tolerance

1. INTRODUCTION

Autonomous robots are being used increasingly in safety-critical environments. These robots need to be reliable to avoid human injuries or material damage, especially when they act outside of an operator's reach. The dynamic and uncertain environments of autonomous robots pose a significant challenge for many low-level actions such as grasping an object or navigating to an exact position. While progress is being made in the accuracy of these low-level actions, avoiding failures altogether seems hardly possible. In an autonomous setting, the robot has to recover from low-level failures by itself to be able to still reach its goal. This need for autonomous failure handling has been recognized in the literature and a multitude of failure handling strategies have been described [1–3].

The focus of this paper is on plan-based robotics, where failure handling strategies are usually part of a manually written robotic plan. Here, failures are treated similarly to programming exceptions, i. e. a low-level module may throw a failure and a higher-level plan has to define handlers to deal with the occurring failures. Failures that are not properly handled will usually cause the robotic plan to crash, stopping the robot entirely.

Ideally, a robotic plan would be written in such a way that all possible low-level failures are handled inside the plan, without any crashes and the need of external interference. With the increasing complexity of robotic plans however, finding unhandled failures is a challenging task.

The typical method of finding unhandled failures are simulations. These are however inherently incomplete and will often not find failures that occur only occasionally. To tackle this problem, we instead propose to use formal methods, in particular symbolic fault injection to find cases in which failures are not properly handled.

Our method is based on the worst-case assumption that any low-level action may fail at any time with any of its possible failure types. We then use symbolic execution to find all cases in which failures are left unhandled.

We use the CRAM Planning Language (CPL) [4] in combination with the symbolic execution engine SEECER [5]. In this work, we extend SEECER to be able to reason about the CPL's failure handling mechanism. We also present a general methodology to implement our worst-case-assumption directly in CPL and present an optimization technique to increase the scalability of our approach. Our method is complete, i. e. it is able to produce either a complete list of all unhandled failures or guarantee that no such failure exists. Since we reason directly on the plan code, we are not limited to certain failure handling strategies.

The remainder of this paper is structured as follows: In the following Section 2 we go over the background necessary for understanding this paper. In Section 3 we present other works related to our approach. Section 4 contains our main contribution, namely the methodology of symbolic fault injection for plan-based robotics. In Section 5 we present experimental results and Section 6 concludes the paper.

The research reported in this paper has been supported by the German Research Foundation DFG, as part of Collaborative Research Center (Sonderforschungsbereich) 1320 *EASE – Everyday Activity Science and Engineering*, University of Bremen (`http://www.ease-crc.org/`). The research was conducted in sub-project P04.

2. PRELIMINARIES

In the following we present relevant background information on the CRAM planning language (Section 2.1) and the SEECER symbolic execution engine (Section 2.2).

2.1. CRAM Planning Language

The *Cognitive Robot Abstract Machine* (CRAM) is a powerful framework for the generation and execution of

```
1  (perform (an action
2                (type searching)
3                (object ?object-designator)
4                (context ?context)
5                (robot-location ?base-location)))
```

Fig. 1.: Performing an action designator in CPL

```
1  (with-failure-handling
2        ((failure-type (e)
3           handler))
4     body)
```

Fig. 2.: CPL failure handling

robotic plans. The *CRAM Planning Language* (CPL) is a core module of CRAM, which allows to define flexible, hierarchical plans for cognitive robots. It is built upon the Common Lisp programming language. The CPL interacts with the environment through means of *action designators*. A designator describes parameters of an action to be executed. In many cases, certain parameters can be omitted. Once the designator is executed, the missing parameters are filled in through different reasoning mechanisms. Designators are executed through the `perform` function.

Example 1. Figure 1 shows a typical call to this function. The keyword `an` builds a designator with the type of designator given as the first argument. In addition to the action designators shown here, there are also designators for objects and locations. The remaining arguments further specify the action. In this case the robot is instructed to search for an object, whose designator is given in `?object-designator`, while standing at the location given in `?base-location`. All other parameters such as the object's likely location are automatically inferred from the context argument.

The CPL provides extensive failure handling capabilities to deal with any unforeseen events caused by the highly dynamic environments of autonomous robots. Failure handling is initiated through the `with-failure-handling` macro. Figure 2 shows the general structure of the macro. The *body* of the macro is executed first. Whenever a failure of type *failure-type* occurs during execution of the body, the *handler* is called. This handler will then try to remove the cause of the failure. Multiple failure handling macros can be used inside each other, creating a hierarchy of failure handlers. When a handler is unable to properly handle a failure, it may choose to ignore the failure, rethrow it to the higher level or throw a new failure of a different type.

2.2. Symbolic verification for CRAM

Formal verification for plan-based robotics is still an emerging field. One approach specifically tailored to CRAM is the symbolic execution engine SEECER. Symbolic execution executes a program while replacing concrete variable values with symbolic variables. During execution symbolic constraints are collected. At certain points in the execution these symbolic constraints are checked for satisfiability by an SMT solver. On branches in the control flow such as an *if* statement, the execution state is duplicated. One execution state continues in the *then* branch and the other in the *else* branch. Due to this duplication and the often complex SMT formulas, symbolic execution has very high runtime and storage space requirements.

SEECER implements symbolic execution for Common Lisp and several CRAM functions and macros. In addition it supports several types of models for the robot's environment. SEECER first compiles the CRAM plan to *CLisp bytecode*. The symbolic execution of the resulting bytecode is based on a stack machine. Each symbolic state is composed of a function stack, a value stack, an additional mapping of variable names to values and a symbolic path condition, i. e. the condition that needs to be met to reach the current execution path.

For more details on symbolic execution for CPL and the SEECER implementation please refer to [5].

3. RELATED WORK

Fault injection has a long tradition in several different application areas to perform robustness evaluations. As such it has been leveraged at the hardware level to induce faults into netlists [6], RTL descriptions [7] or even system-level models [8] as well as at the software level [9].

Symbolic fault injection [10] extends upon the idea of traditional fault injection by using non-deterministic locations for the injection and hence enables to produce complete coverage of the system under verification with guarantees regarding its robustness. As such symbolic fault injection is a very powerful technique for finding gaps in the failure handling of complex systems. Symbolic fault injection has been mostly applied to embedded systems [10, 11]. To the best of our knowledge this paper is the first to use symbolic fault injection for plan-based robotics.

Other approaches to formal verification of robotic plans usually require a complex specification of the robots environment, either explicitly or through safety properties that describe the robots desired behavior. In [12] an environment model is constructed using the Discrete Event Calculus. This allows for the verification of general properties, but the environment modeling requires a lot of manual work and is error-prone.

Often the robotic plan is formulated in a logical formalism as well, e. g. in temporal logics [13–15], petri nets [14] or Golog [16]. These allow to use existing reasoning techniques, but they also limit the expressiveness and practicability of the robotic plan.

This paper uses the robotic planning language CPL instead, which does not suffer from the strict framework and reduced expressiveness of these formalisms. Instead, CPL is Turing-complete and can be directly executed.

4. SYMBOLIC FAULT INJECTION FOR CRAM

This section describes our approach on fault injection for CPL plans. We start with an overview of our approach in the following section.

```
1  (with-failure-handling
2      ((failure-type (e)
3          handler))
4    body)
5
6
7
```

(a) Original code

```
1  (defun (e) handler1
2      (if (typep (type-of e) failure-type)
3          handler
4          (rethrow-failure e)))
5  (start-failure-handling "HANDLER1")
6  body
7  (end-failure-handling)
```

(b) Rewritten code

Fig. 3.: Rewriting scheme for failure handlers

4.1. Overview

Our approach takes a CPL plan and tries to find top-level failures that can occur in it. Most failures are thrown inside very low-level modules that are responsible for the execution of mechanical actions such as grasping. We want to abstract from the internals of these modules, since they often depend on the current state of the environment. Instead, we consider all types of failures that may be thrown by the module and replace the concrete implementation of the module with the worst-case assumption that any of those failure types may occur whenever the module is called.

We call the actions for which we apply this assumption *atomic actions*. Please note that the notion of atomic actions is flexible, i. e. depending on the desired accuracy and runtime, a user could choose higher or lower cut-off points.

The core of our approach is the symbolic execution engine SEECER. We extended SEECER to support the extensive failure handling capabilities that are present in the CPL. The details of this extension are described in Section 4.2.

Our fault injection approach is based on the general worst-case assumption that any action may fail at any time with any of its possible failure types. To reflect this behavior, we built a general environment model that implements this assumption for all atomic actions of the CPL. In addition we implemented a similar worst-case assumption for all reasoning subroutines. The details of this environment model and the reasoning subroutines are presented in Section 4.3.

Finally, we propose an additional optimization technique to reduce the runtime of our approach in Section 4.4.

4.2. Extending SEECER with failure handling

Prior to this work, SEECER supported the core of the CPL as well as a multitude of Common Lisp functions. To enable symbolic fault injection, the failure handling functionalities of the CPL were implemented into SEECER.

The nested Common Lisp code is generally problematic for a symbolic execution engine. Therefore, as a first step, the nested failure handling macro is rewritten into a more sequential control flow. The applied rewriting is illustrated in Figure 3. Figure 3a shows the code before and Figure 3b after our rewriting scheme is applied. As shown, the failure handler is placed inside a new function and guarded by the condition that the type of the active failure is equal to or inherited from the *failure-type* (Line 2). Otherwise the handler is not executed, but instead the failure is re-thrown to be handled by a higher-level handler (Line 4). We call this new function the *handler function* to differentiate the whole function from the original handler that is now part of it.

The body is encapsulated inside two new functions *start-failure-handling* (Line 5) and *end-failure-handling* (Line 7). SEECER uses these functions to manage failures and handler functions internally. Information about handler functions are organized as a stack, with start-failure-handling pushing a new handler function onto the stack and end-failure-handling removing the top function. When a failure occurs, the execution jumps to the newest handler function on the stack. This is done by pushing a new element onto the function stack similar to a normal function call. The handler function is then executed. If the same or a new failure is thrown inside the handler function itself, the next handler from the stack is executed. If no failure is thrown inside the handler, the failure has been successfully handled and execution jumps back to the return address, i. e. the line after which the failure was initially thrown. If a failure is thrown with an empty handler stack, the failure reaches the top level of execution. We call these failures *top-level failures*. In a normal execution they lead to a crash of the plan. SEECER can be configured to either terminate the whole execution or just the current context in this situation. In the first case only the first top-level failure would be reported and in the second case all top-level failures would be collected before SEECER terminates. The user can decide between the two modes depending on their needs. If no top-level failure occurs at all, the plan's failure handling is proven to be complete.

4.3. Symbolic Substitution of Atomic Actions

In this paper we want to use the failure handling described in the previous section to find all possible top-level failures. This is based on the worst case assumption that any atomic action may fail at any time with any of its possible failures.

The worst-case assumption is completely implemented into Common Lisp as follows. For each execution of an action, a new symbolic integer is created. Each of the n possible failures is then assigned a unique integer between 1 and n. A chain of `if` statements now ensures that each failure is thrown iff the symbolic integer has that failure's respective value. For all other values no failure is thrown.

Example 2. Consider an action of type *Grasping*, that tries to grasp an object. There are two possible types of failure that may occur. If the robot's motion planning module does not find a sequence of motions to reach the object – usually because it is too far away – a failure of type `gripper-goal-not-reached` is thrown. If the robot instead tries to grasp the object, but detects that its grippers have fully closed (and

```
1  (let* ((sym-ctr (sym-int symbolic-name)))
2      (if (= 1 sym-ctr)
3          (cram-failure (make-instance
4            'gripper-goal-not-reached))
5          (when (= 2 sym-ctr)
6              (cram-failure (make-instance
7                'gripper-closed-completely)))))
```

Fig. 4.: Implementation of the worst case assumption for the Grasping action

```
1  (defun handler1 (failure)
2      (if (typep (type-of failure)
3            'gripper-goal-not-reached)
4          (print "WARNING: Grasping failed")
5          (rethrow-failure failure)))
6  (start-failure-handling "HANDLER1")
7  (perform
8      (an action
9          (type grasping)
10         (object obj)))
11 (end-failure-handling)
```

Fig. 5.: A failure handler without side effects

therefore failed to grasp the object), a failure of type `gripper-closed-completely` is thrown. Figure 4 shows the implementation for the Grasping action. In Line 1 a new symbolic integer is created. The variable `symbolic-name` contains a string for the internal naming of the variable in the SMT solver. `sym-ctr` now contains the variable itself. If the variable is equal to 1, a `gripper-goal-not-reached` failure is created and thrown in Line 4. Similarly, if it is equal to 2, Line 7 throws a `gripper-closed-completely` failure. For all other values, no failure is thrown.

Please note that the assignment of integer values to failures is arbitrary. A different assignment will still produce the same outcome as long as all failures are assigned at least one unique value.

In addition to atomic actions, the environment model also needs to deal with complex reasoning subroutines that are part of many high-level planning frameworks like CRAM. These routines are often complex and take into account the current environment, the robots belief state and dynamically changing knowledge bases. Following our worst-case assumption, we have to reason about all possible results from the reasoning subroutines. Similarly to the atomic actions this is realized through a pure Common Lisp implementation. For each call of a reasoning subroutine, a symbolic variable of the respective type is created. The value may be restricted through `assume` statements (e. g. to a certain numerical range) and is then returned.

Such a general methodology as presented here will often create a large number of symbolic values and result in a large number of symbolic paths. Therefore, the next section will discuss a technique to reduce the search space through means of state caching.

4.4. State Caching

In many cases a failure handler will have little to no side effects, i. e. the state after the handler has finished is identical to the state in which no failure occurred in the first place.

Example 3. Consider the plan excerpt in Figure 5. As shown in the previous example, there are three possible outcomes of the grasping action performed in Line 7: The action may throw a `gripper-goal-not-reached` failure, a `gripper-closed-completely` failure or no failure at all. In the first case, the handler would print a warning and then jump back to Line 11 and in the third case the handler would not be called at all and instead the execution would continue as normal, also with Line 11. Since the handler has no side effects at all, both execution states would be identical.

Since both states are identical, they will lead to the same execution traces upon further symbolic execution. Because of this it is safely possible to only continue execution on one of the states without affecting the final result. This is an instance of *state merging* [17], an established optimization technique in symbolic execution. Usually state merging will compare symbolic states and if two states are similar by some metric, their path conditions and variable assignments will be combined. This combination results in less, but more complex symbolic states, shifting complexity from the search algorithm to the SMT solver. For an effective use of state merging, a large number of symbolic states must be active at the same time. As demonstrated above, states during fault injection for typical CRAM plans are not only similar, but identical. This enables us to use a simpler version of state merging with less overhead which we call *state caching*. State caching only acts on identical states at certain manually chosen checkpoints in the plan. At the checkpoints the states are stored inside a cache. Whenever a state s is identical to a previously stored state s', the current execution trace can be terminated, since all results after state s have already been produced after s'. This way identical states are only processed once.

The comparison between states is based on the states' function stack, value stack, variable assignments and path condition. These also include values that have been used prior to the current point in the execution, but are not relevant to any decisions after that point. This is especially true for the symbolic values introduced in the last section as they are used only once immediately after being created. These values can therefore be ignored when it comes to comparing two states.

Example 4. Consider again the previous two examples. Since the symbolic variable `sym-ctr` is used to differentiate between the type of failure that is thrown, the two states from the previous example will have differing path conditions. The state which throws the `gripper-goal-not-reached` failure, will have $\texttt{symctr} = 1$ in its path condition. The state which did not throw a failure will instead have a path condition of $\texttt{sym-ctr} \neq 1 \wedge \texttt{sym-ctr} \neq 2$. Since `sym-ctr` is not used in any future paths of the program, this difference may however be ignored, making the two states identical again.

Currently the values to ignore are determined manually, but for future work we plan to do this automatically based on the plan's control flow.

We implemented the state cache in an efficient structure using hash values for fast comparisons. These hash values

are constructed by first hashing the individual entries in the function stack, value stack, variable assignment and path condition and then combining all entries via the XOR function.

5. EXPERIMENTAL EVALUATION

This section presents our experimental evaluation. Our main research questions are whether symbolic fault injection is suited for plan-based robotics and how fast our approach is for typical CRAM plans. We investigate a system of CRAM plans for generalized fetch and deliver actions. These plans are described in more detail in Section 5.1. The final results and their interpretation with regard to our research questions are presented in Section 5.2.

5.1. Robotic Plans and Actions

We evaluate our approach on a system of generalized fetch and deliver plans. The plans implement different subroutines that are used to transport objects from one place to another, including searching for objects and opening and closing containers. The plans are very general, i. e. they are independent of concrete objects or locations. They can be roughly grouped into three classes: atomic actions (e. g. Setting-gripper), low-level plans (e. g. Picking-up) and high-level plans (e. g. Fetching). Here the low-level plans use atomic actions internally and high-level plans use atomic actions and low-level plans. One high-level plan, *Transporting* also uses several other high-level plans, making it the most complex plan of the system.

Both the low-level and high-level plans are equipped with several failure handlers. These are often organized hierarchically, i. e. when one handler is unable to deal with the underlying problem, it throws a new failure which is handled by a higher-level handler. The deepest handler hierarchies are found in the Searching and Fetching plan with 5 nested failure handlers each.

There are a total of 17 atomic actions, 4 low-level and 8 high-level plans. Each atomic action has between 1 and 3 possible failures with some overlap between actions, for a total number of 13 distinct failure types. The high-level plans have a differing number of arguments, ranging from 1 to 6. There are also 7 reasoning subroutines which decide on certain arguments for some of the plans. All plans combined amount to 2284 lines of bytecode.

5.2. Experimental Results

We substituted all low-level actions and reasoning subroutines according to our approach in Section 4.3. We then executed our extended version of SEECER on the resulting code. SEECER was configured to report all top-level failures and not terminate after the first find. We evaluated each high-level plan on its own, implicitly also covering all low-level plans and atomic actions. In addition, the arguments for each high-level plan were kept fully symbolically, considerably adding to the complexity. All experiments were conducted on a Linux machine running an Intel Core i5-7200U CPU with 2.50 GHz clock rate.

We found unhandled top-level failures in all eight plans. Some of these would have been easy to find without formal methods as well, e. g. when handlers for certain failures were simply missing. Other failures would be lot harder to spot manually or via simulation-based testing. For instance, some top-level failures only occurred when an action inside a failure handler itself also failed. There were also cases where several handlers were unable to properly handle a failure until it reached the top level. Our first research question has therefore been answered positively. Our evaluation strongly suggests that symbolic fault injection is a well-suited tool for plan-based robotics. This leaves only the runtime question to be answered.

Table 1.: Experimental results on the high-level plans

Plan	With state caching		Without state caching	
	#paths	time	#paths	time
Navigating	7	11s	7	11s
Turning	17	12s	24	12s
Searching	55	12s	877	12s
Delivering	373	14s	47185	126s
Accessing	1841	18s	7121	22s
Sealing	1841	18s	7121	22s
Fetching	4105	37s		timeout
Transporting	59161	568s		timeout

Table 1 summarizes the runtime results of our evaluation. We report the number of symbolic paths and the total runtime for all eight high-level plans. To show the effect of our proposed state caching technique, we report the results both with and without state caching enabled. The columns of Table 1 report (from left to right) the plan under verification, the number of symbolic paths with state caching enabled, the runtime with state caching enabled and then both metrics when state caching was disabled.

As expected, the runtime for each plan correlates with the number of symbolic paths that are explored. Both metrics depend on the underlying complexity of the plans. Simpler plans such as Navigating lead to only few paths. The majority of its $11s$ runtime are not even used for symbolic execution, but rather for the initial setup and compilation of the plan into bytecode. The Transporting plan on the other hand uses all other high- and low-level plans. Therefore it is the most complex out of all evaluated plans by far. This becomes apparent in the large number of symbolic paths even with state caching enabled. Nonetheless, its runtime is still below 10 minutes, which is perfectly acceptable for a complete analysis with all possible arguments and a complete list of top-level failures. The analysis for all other plans was finished in under a minute. The two rightmost columns of the table show the effect of disabling our state caching technique. The influence of state caching is clearly visible when looking at the Delivering, Fetching and Transporting plan. Without the optimization, the number of paths of the Delivering plan increased by a factor of over 100 which lead to a runtime increase of 900%. The Fetching and Transporting plans did not finish within the 1 hour time limit, which means a runtime increase of at least 9729% for the Fetching plan. For the other plans, the effects of state caching were less apparent, but nonetheless always positive or at least neu-

tral. We did not observe a case were the slight overhead of state caching actually impacted the runtime negatively.

Overall, our proposed approach was able to completely analyze typical CRAM plans within a short time, for most plans within less than a minute. For the more complex plans this is primarily thanks to our proposed state caching technique.

6. CONCLUSION

Many plan-based robotic systems use failure handling to deal with low-level failures caused by the dynamic and uncertain environment. With the increasing complexity of robotic plans, a plans failure handling will often be incomplete or contain errors. In this paper we proposed a methodology to automatically find failures that reach the top-level without being handled or prove their absence. We presented an extension to the symbolic execution engine SEECER to incorporate failure handling. Our approach is based on the worst-case assumption that any action may fail at any time with any of its possible failure types. We implemented this assumption in Common Lisp for a straightforward integration with SEECER. Finally we presented state caching as an optimization technique to deal with the large amount of symbolic states. Our experiments show that symbolic fault injection is an effective technique for plan-based robotics. We were able to find unhandled failures in several CRAM plans. Thanks to our proposed state caching technique, the runtime was below one minute for all but one plan.

Our current implementation is mostly automatic, but requires some manual preparation of the CPL plan under verification. Especially our state caching technique requires the developer to chose effective checkpoints and variables that can be safely ignored. For future work, we plan to fully automate this preparation through analysis of the underlying control flow graph.

REFERENCES

[1] R. R. Murphy and D. Hershberger, "Handling sensing failures in autonomous mobile robots," *The International Journal of Robotics Research*, 1999.

[2] T. Lienert, L. Stigler, and J. Fottner, "Failure-handling strategies for mobile robots in automated warehouses," in *33rd INTERNATIONAL ECMS Conference on Modelling and Simulation*, 2019.

[3] G. Kazhoyan, S. Stelter, F. K. Kenfack, S. Koralewski, and M. Beetz, "The robot household marathon experiment," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021.

[4] M. Beetz, L. Mösenlechner, and M. Tenorth, "Cram—a cognitive robot abstract machine for everyday manipulation in human environments," in *Intelligent Robots and Systems*, 2010.

[5] T. Meywerk, M. Walter, V. Herdt, D. Große, and R. Drechsler, "Towards Formal Verification of Plans for Cognition-enabled Autonomous Robotic Agents," in *Euromicro Conference on Digital System Design (DSD)*, 2019.

[6] S. Aftabjahani and Z. Navabi, "Functional fault simulation of vhdl gate level models," in *Proceedings VHDL International Users' Forum. Fall Conference*, 1997.

[7] P. Thaker, V. Agrawal, and M. Zaghloul, "Register-transfer level fault modeling and test evaluation techniques for vlsi circuits," in *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*, 2000.

[8] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Fault-effect analysis on system-level hardware modeling using virtual prototypes," in *2016 Forum on Specification and Design Languages (FDL)*, 2016.

[9] M. Kooli, A. Bosio, P. Benoit, and L. Torres, "Software testing and software fault injection," in *2015 10th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2015.

[10] D. Larsson and R. Hähnle, "Symbolic fault injection," in *International Verification Workshop (VERIFY)*, 2007.

[11] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Symplfied: Symbolic program-level fault injection and error detection framework," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008.

[12] T. Meywerk, M. Walter, V. Herdt, J. Kleinekathöfer, D. Große, and R. Drechsler, "Verifying safety properties of robotic plans operating in real-world environments via logic-based environment modeling," in *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, ser. Lecture Notes in Computer Science, 2020.

[13] F. Bacchus and F. Kabanza, "Using temporal logics to express search control knowledge for planning," *Artif. Intell.*, 2000.

[14] B. Lacerda, "Supervision of discrete event systems based on temporal logic specifications," Ph.D. dissertation, 2013.

[15] L. Lindemann, G. J. Pappas, and D. V. Dimarogonas, "Reactive and risk-aware control for signal temporal logic," 2021.

[16] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, "Golog: A logic programming language for dynamic domains," *The Journal of Logic Programming*, 1997.

[17] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.