Using Virtual Prototypes for Causal Fault Explanation at System Level *

Caroline Dominik[1], Rolf Drechsler [1,2]
[1]Institute of Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{cardom, drechsler}@uni-bremen.de

Abstract

With *Virtual Prototype*s (VPs), it is possible to significantly improve the debugging process during system design as the interaction of components can be analyzed and internal variable values can be accessed. But in case of detecting a fault, it remains challenging to determine its specific cause, as the load of available information can be overwhelming.
To address this issue, we propose to monitor a VP at runtime to extract an under-approximating formal model of the system behavior, and derive causal explanations by model checking. For this, we determine the sufficient condition of a failure based on the order and context of events. This is demonstrated using an abstracted controller for a wind turbine, implemented using a RISC-V VP.

1 Introduction

The development of *Cyber-physical system*s (CPSs) is a complex task, as they are the union of several components. These elements can be different sensors for measuring the physical world, computational elements and actuators for translating digital orders into physical motion. A design flow based on *Virtual Prototype*s (VPs) is an established approach to support this task [1]. By simulating the involved *Hardware* (HW) components, insight into their internal signals is granted. Further, based on the holistic system view of VPs, the interconnection of all modules can be inspected early on. Both aspects are valuable during the development of CPSs, as the early detection of errors becomes possible, which supports the debugging process. But the detection of a fault alone does not always give any insight into how to fix it. Consider for example *Bounded Model Checking* (BMC) [2]: A formal model is checked against its specification and a counterexample is produced in case a property is violated. While this counterexample does trigger the erroneous behavior, it may not be apparent which exact event or sequence of events is the actual cause. Even though it has minimal length, some parts of the counterexample are usually not directly connected with the error. This can, e.g., include initialization statements, or recurring actions which appear in most traces like the events required for processing any input. Hence, the counterexample can be misleading when a designer wants to identify the necessary steps to fixing the fault. This is because the counterexample is merely a symptom of the underlying fault. Further explanation is needed.

Self-explanation of systems has emerged as a new system property [3] [4] [5] [6] to improve their reliability and the overall trust in digital systems. This includes several possible targets for explanations, such as finding the cause for a certain execution time or for the decision of a control unit. It also includes the explanation of faults. Compared to classical debugging, **debugging based on fault explanation** requires less manual analysis of the system since the root cause for the failure is given. This reduces the time necessary for fixing errors and increases the trust in the system, as its behavior is better understood. To identify a cause, e.g., if BMC fails, it is necessary to argue about the general behavior of a system instead of focusing on the specific events of a single counterexample. It is the goal to determine the sufficient condition for the failure. Following this concept, a counterexample given by BMC was reduced to a *fault seed* in [7]. By extracting the events of the counterexample which are the sufficient condition for the fault to be triggered, only the information that is necessary to understand the cause is kept.

In this paper, we take a first step towards *Self-Explainable Virtual Prototype*s (SXVPs) by applying such causal explanation based on fault seeds during a VP-based design flow. This further increases the support that VPs provide for system design. This requires a formal model representing the behavior of the entire system. As this is often missing in realistic development settings, runtime monitoring for model extraction at system level is proposed as an approach. But a central challenge with formal models is the possible state space explosion, which significantly effects the performance of any methods, such as BMC, being applied to them. Hence, it is necessary to restrict the formal model in some way to enable applying causal explanation. For this, a high-level monitoring of the VP is chosen in this approach. The central *Software* (SW) variables and inputs and outputs of registers are observed, while internal information, such as register values, are ignored. To further limit the increase in required resources, an under-approximating model is used.

This paper is structured as follows. First, the explanation framework is introduced by sketching the workflow of monitoring at system level and by reviewing fault explanation based on fault seeds. Further, the fault seeds are extended to be context-dependent. Then, the explanation approach is applied to an abstracted wind turbine controller and the violation of two safety properties. This is implemented using a

*Funded by Deutsche Forschungsgemeinschaft (DFG) GRK 2972 "CAUSE" - Project number: 513623283.

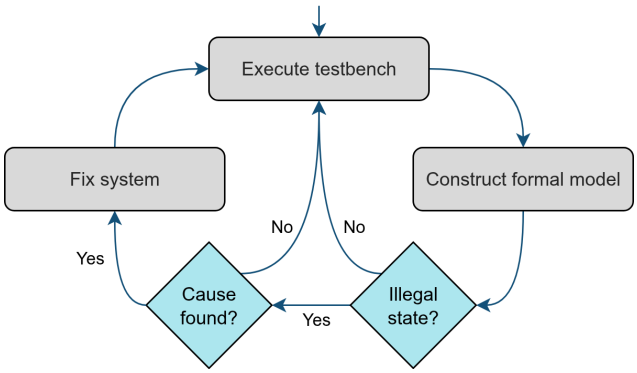

Figure 1: Workflow of fault explanation at system level.

VP for RISC-V. Finally, the approach is discussed w.r.t. the effect of model completeness, the quality of explanation, its scalability and robustness and its abstraction level.

2 Explanation Framework

In the following, the proposed framework for causal fault explanation is presented. The underlying workflow is shown in Fig. 1.

2.1 Monitoring at System Level

Describing the behavior of an entire system that consists of several components with a formal model requires computing the product automaton of all its models. As explained in Chapter 4 of [8], the resulting number of states is the product of the size of all models. This can significantly increase the complexity of computing the model or of applying any method to it. This is not useful in the considered scenario, as many of the states in the product automaton are not reachable.

Hence, to obtain a model for the system level while reducing the required resources, we propose computing an under-approximation of a product automaton of the models for the SW and each HW peripheral. This model is computed based on the execution of several testbenches, as depicted in Fig. 1. A monitor observes the SW and each HW peripheral at runtime and extends the model for each iteration. That way, the model is successively refined by each testbench. This is done without computing any unreachable states and without ever having to actually apply the resource-intensive operator for calculating a product automaton.

2.2 Causal Fault Explanation

Let the tuple $M = (S, S_0, R, L)$ be a Kripke structure (Chapter 3 of [8]) that formally defines a system. For this, S is a finite set of states, $S_0 \subseteq S$ is a finite set of initial states and $R \subseteq S \times S$ defines the transition relation. Let AP be a set of atomic propositions, so that $L : S \to 2^{AP}$ is a labelling function which maps each state to those propositions which hold in that state. The system specification $Spec$ is given by *Linear Temporal Logic* (LTL) properties over AP. Failed BMC for M and S produces a counterexample ce. ce is defined as a finite sequence of states $(s_0, s_1, \ldots, s_n)$ with $s_0 \in S_0$ and $\forall 0 \leq i < n : s_{i+1} = R(s_i)$, so that $Spec$ is violated. As mentioned before, ce usually cannot serve as an explanation by itself. It is necessary to isolate the relevant aspects of the produced witness to determine the cause of the fault.

2.2.1 Explanation based on Fault Seeds

This is addressed in [7]. In their approach, they consider the event labels σ of transitions over an event set Σ. This is expressed by a Kripke structure by extending it to $M = (X, S_0, R_\Sigma, L_\Sigma)$ using $X = S \times (\Sigma \cup \{e\})$ with null event e and $R_\Sigma \subseteq X \times X$. The labelling function is $L_\Sigma : X \to AP_\Sigma$ with $AP_\Sigma = AP \cup \Sigma$, so that $\forall (s, \sigma) \in X : L_\Sigma(s, \sigma) = L(s) \cup \{\sigma\}$. With this, the event labels are moved from each transition to the destination state of that transition. According to that, a counterexample ce is mapped to the corresponding finite sequence of events $(\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$. The authors propose to derive an event trace $(\sigma_0, \ldots, \sigma_k)$ based on the events of ce called *fault seed*. This fault seed is the minimal sufficient condition for a fault to appear. This means, the events $(\sigma_0, \ldots, \sigma_k)$ cannot be executed without the fault appearing afterwards. So the system must satisfy $fault\ seed \Rightarrow \neg Spec$.

To compute the fault seed, all possible fault seed candidates with length k are iterated for $k = 1$ until $k = |ce|$ with $|ce|$ being the length of the counterexample. The candidates include all possible combinations of events in ce. Each candidate is formulated as an LTL property f, to check the following:

1. f can actually be executed on M. For this, a path in M must exist, which executes the events of f in the order defined by f.
2. $M \models (f \Rightarrow \neg Spec)$ holds, meaning the execution of the candidate implies the violation of the specification.

To check these two statements, *Boolean Satisfiability Problem* (SAT)-based BMC is applied. If both hold, a fault seed was found and is returned.

2.2.2 Context-dependent Fault Seeds

But a cause cannot be expressed by such a fault seed, whenever an event has to appear in a certain context to trigger the fault. Hence, we propose to extend the approach of [7] by using **context-dependent events** defined by the tuple (c, σ). These include a state restriction c next to the event restriction σ. The state restrictions constrain the variable values of the origin state of a transition, while the event restrictions constrain which event has to happen by traversing using a certain transition. They are included in the candidates, if no fault seed with only context-dependent events can be found for the current k. This further allows the computation of a **cause-effect chain**. For this, the fault seed for the context restriction of the last fault seed is calculated. This is done by computing the fault seed for the safety property "$G\ \neg c$". This is repeated, as long as the resulting fault seed contains a context-dependent event or as long as no (new) seed is found.

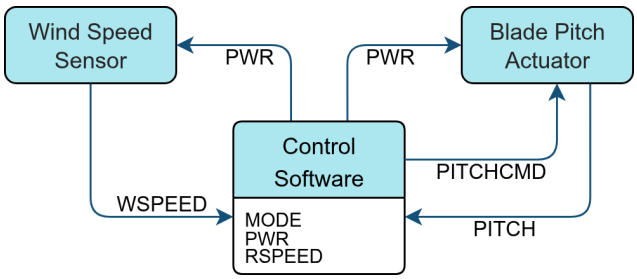

Figure 2: System components of an abstracted wind turbine controller.

In the proposed explanation approach in Fig. 1, the causal fault explanation is triggered if a property violation is found by applying SAT-based BMC after executing a testbench. The cause is calculated to guide fixing the error in the system before the next iteration. In case no cause is found or the explanation is not sufficient, the model is extended, so that different cases in the model become more distinguishable.

3 An Illustrative Example

To illustrate the proposed approach, an abstracted wind turbine controller is used, as explained in the following. This example case is implemented using the open-source RISC-V VP [9] [1].

3.1 System Specification

Since the interaction of components is of interest here, the system consists of three components, as depicted in Fig. 2: The *control SW* runs on the RISC-V processor and switches between control modes $MODE \in \{0,1,2,3\}$ for stopped, boot, operation and shutdown, respectively. To simplify computation, the power variable PWR is not an input but internally set during boot mode. Then, the controller activates both peripherals by setting their PWR inputs. All variables are reset during turnoff, respectively. During operation, a *wind speed sensor* peripheral measures the current wind speed $WSPEED \in \{0,1,2,3,4\}$ and outputs it. Based on $WSPEED$, the control SW sets the desired blade pitch angle $PITCHCMD$ to 2, 1, or 0 for full, reduced or no wind exploitation, respectively. As can be seen in the first two columns of Table 1, full exploitation is only used for slower wind speeds $WSPEED \in \{2,3\}$. To prevent damage and to not exceed the limit of generated power, reduced exploitation is used for $WSPEED = 3$. The turbine is stopped in case of no wind and for storms, so $WSPEED \in \{0,4\}$. A *blade pitch actuator* peripheral executes the pitch command and sends back the actual pitch $PITCH \in \{0,1,2\}$. The current turning speed of the rotor $RSPEED \in \{0,1,2,3,4\}$ is part of the controller in this example. In Table 2, the calculation of $RSPEED$ is shown in the second column w.r.t. the three settings of $PITCH$ in the first column. $RSPEED$ is 0 for no wind exploitation, is limited to 2 but otherwise equal to $WSPEED$ for reduced wind exploitation and is equal to $WSPEED$ for full wind exploitation. The resulting value of $RSPEED$ w.r.t. each possible wind speed can be seen in the last column of Table 1. In the RISC-V VP, the wind speed sensor and the blade pitch actuator are implemented as peripherals, communicating via *Transaction Level Modeling* (TLM). Their inputs and outputs are implemented as memory mapped registers.

[1] https://github.com/agra-uni-bremen/riscv-vp

WSPEED	PITCH	RSPEED
0 (= *no*)	0	0
1 (= *slow*)	2	1
2 (= *medium*)	2	2
3 (= *fast*)	1	2
4 (= *storm*)	0	0

Table 1: Wind turbine's specified behavior of blade pitch and rotor speed w.r.t. the wind speed.

3.2 Model Extraction

In this paper, the aim of explanation is the system level, hence faults occurring in the interaction of several system components. Therefore, we propose a model abstraction with a focus on the interfaces of components instead of VP internals. For this, the inputs and outputs of peripherals and the internal variables of the control SW are monitored. A state of the model is then given by any assignment of those variables occurring at runtime. Events are given by the changes applied to the variables, which result in the transition to a different state. Internal details of the VP like the timer and registers are abstracted, as they would massively increase the state space without offering new insights into the interaction of components.

To obtain this model, a monitor is added to the RISC-V VP, as depicted in Fig. 3. Each peripheral has direct access to it, to update the current state whenever the value of a register is altered. Direct access by SW to any internals of the VP is not possible. Therefore, the monitor has TLM registers, which the SW uses to update the monitor on changes of the SW variables. The self-explaining monitor can then be used be a designer to generate causal explanations of occurring faults. By this, the designer becomes the addressee of the explanation.

3.3 Computing the Cause

To demonstrate the computation of a cause, two functional safety properties are specified to define $Spec$ for the run-

PITCH	RSPEED
0 (= *no*)	0
1 (= *reduced*)	$(WSPEED < 2)\,?\,WSPEED : 2$
2 (= *full*)	$WSPEED$

Table 2: Wind turbine's specified behavior of rotor speed w.r.t. the blade pitch.

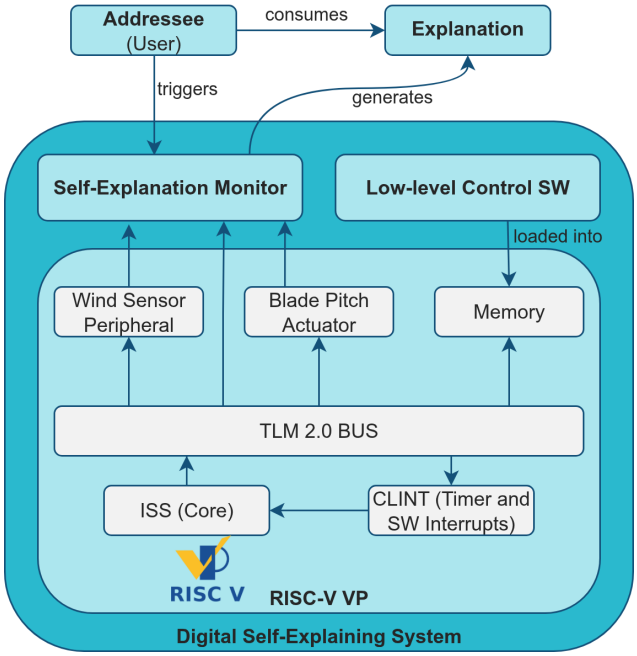

Figure 3: Components of RISC-V VP implementation of fault explanation for a wind turbine.

ning example and their violation is analyzed by using the concepts shown in Section 2.2.

3.3.1 Property 1

The safety property "$G\ \neg(RSPEED\ ==\ 4)$" avoids that the rotor's speed limit is exceeded. This ensures that no parts of the wind turbine can take damage. This property is, e.g., violated using the input sequences $(0, 1, 0, 1, 2, 0)$ and $(0, 1, 2, 4, 2)$ for the value $WSPEED$ of the wind speed sensor. This creates a model of 47 states. By applying the algorithm described in Section 2.2, the fault seed

$$(WSPEED = 4)$$

is computed. This identifies high wind speed as the cause for the faulty behavior. While this gives some insight into the system's behavior, it is useful to execute another testbench to obtain a more differentiated result. After extending the model by executing input sequence $(0, 1, 2, \mathbf{3}, 4, 2)$, 6 states are added. The resulting formal model can be seen in Fig. 4. Per state, a state ID and the values of the variables introduced in Section 3.1 are depicted. For this, the variables are named in combination with an abbreviation of their component. *WSS* is used for the wind speed sensor peripheral, *BPA* for the blade pitch actuator and *main* for the control SW. To increase readability, the sequences for boot and shutdown were summarized each in a single state. The detected illegal state is highlighted in red, the states of the respective counterexample are highlighted in light red. It can be seen, that based on this counterexample, roughly a third of the depicted states are highlighted. This shows, that a counterexample by itself is not well suited for identifying a specific event which is the cause for reaching the detected illegal state. When applying the algorithm for computing the cause now, based on only context-independent events

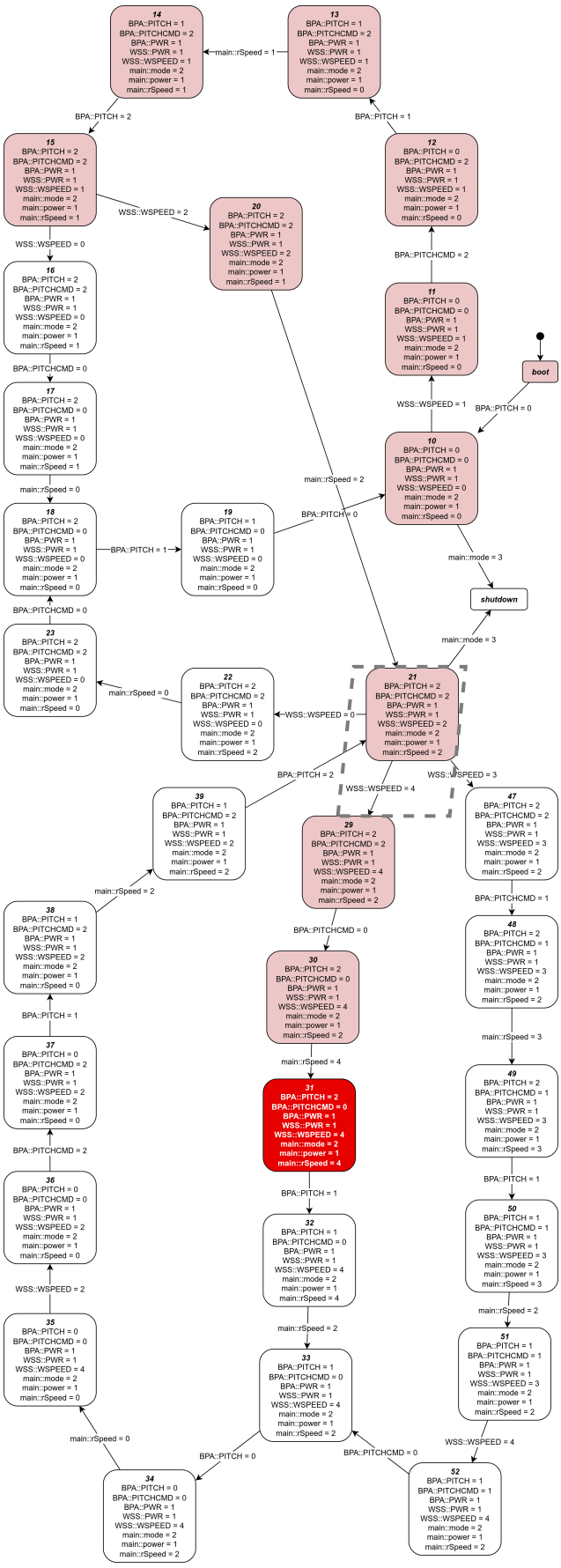

Figure 4: Formal model with violation of Property 1.

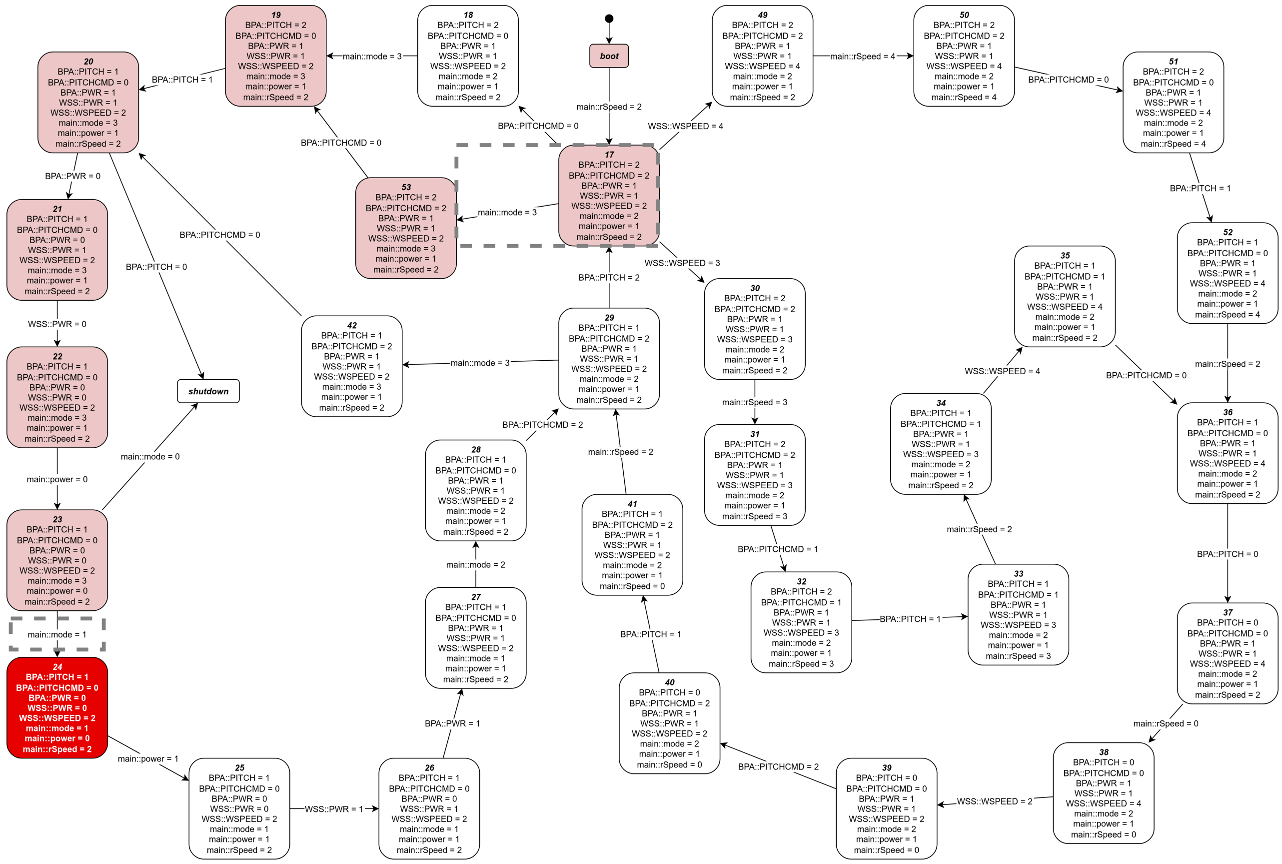

Figure 5: Formal model with violation of Property 2.

no fault seed can be found. By including context-dependent events the resulting fault seeds with $k = 1$ are

- $(PITCHCMD == 2, WSPEED = 4)$,
- $(PITCH == 2, WSPEED = 4)$,
- and $(WSPEED == 2, WSPEED = 4)$.

The corresponding state and transition are highlighted by a gray, dashed border in Fig. 4. Hence, the error is caused by a fast increase in wind speed of $(WSPEED = 2)$ followed by $(WSPEED = 4)$ with no $(WSPEED = 3)$ in between, which cannot be expressed by context-independent events only. This implies that adjusting the blade pitch angle takes too long to react to such sudden shifts in wind speed. This additional information is crucial for fixing the fault, as it shifts the designer's attention to this corner case. Possible solutions could be to use reduced wind exploitation for $(WSPEED = 2)$ or to restrict the input sequences to not include such jumps in values if this is not representative for actual wind behavior. Further, the computation of a cause-effect chain can be demonstrated. When selecting, e.g., $(PITCH == 2, WSPEED = 4)$ to compute the cause-effect chain, the fault seed for the property "$G\ \neg(PITCH == 2)$" has to be computed. This gives the fault seed $(PITCHMCD = 2)$ for $k = 1$, which further clarifies the order of events causing the error.

3.3.2 Property 2

Property "$G\ ((MODE == 1) \rightarrow (PITCH == 0))$" ensures that the blade pitch is not set during the boot phase of the system. To detect a violation of this property, a restart is added to the input sequences. The testbenches $(0, 1, 2(R), 3, 4, 2)$ and $(0, 1, 2(R), 4, 2)$ now both include a restart in the third time step while $WSPEED$ is set to 2. The resulting formal model contains 55 states and can be seen in Fig. 5. Again, the boot and shutdown traces are summarized as single states. An illegal state is highlighted in red and the respective states of the counterexample are highlighted in light red. With $k = 1$ no fault seeds are found. With $k = 2$ the fault seed

$$(PITCHCMD == 2, MODE = 3), (MODE = 1)$$

is computed. The corresponding state and transitions are highlighted by gray, dashed borders in Fig. 5. The fault seed shows that a shutdown during full wind exploitation, followed by a reboot is the causal explanation of the fault. This implies that the duration of resetting the blade pitch takes longer than the shutdown, which prevents a correct reset of variables during the shutdown phase. The designer can use this information to fix the system by, e.g., only allowing to exit the shutdown mode after all variables are reset correctly.

The computed fault seed shows that both order and context of events can be necessary for expressing a causal relation. Based on only context-independent events, the shut-

down during full wind exploitation cannot be expressed. Compared to the fault seed above, the context-independent fault seed

$$(PITCHCMD = 2), (MODE = 3), (MODE = 1)$$

does not exclude system executions with, e.g., a $(PITCHCMD = 1)$ between the events $(PITCHCMD = 2)$ and $(MODE = 3)$. In the same way, several events are necessary to express the fault, as the order of both events is a central aspect of the explanation and a single context-dependent event couldn't express this.

4 Discussion

After illustrating the proposed framework based on the example of a wind turbine controller, its advantages and disadvantages can be analyzed. This is done for the completeness of the extracted model, for the quality of the given causal explanations, for the scalability and robustness of the approach and, finally, for the chosen abstraction level.

4.1 Model Completeness

The proposed framework for causal fault explanation is based on an under-approximation of the system model, which can result in missed causes. Therefore, different types of application for the framework are discussed in the following:

- **Random testing:** This approach is used in this paper. As depicted in Fig. 1, a batch of testbenches is computed until an error occurs. If the current under-approximated model is not sufficient for computing a cause, more testbenches are executed until a cause is found.

  The focus of this approach is achieving quick results without using additional resources for generating inputs for the system. Further, by under-approximating the model, the execution time for any algorithm that is applied to the model is reduced, which speeds up the detection of causes. While this can detect many faults and their causes, there is no guarantee for the discovery of each fault and a successful cause computation, especially for corner cases or more complex explanations.

- **Guided testing:** The approach can be extended by guided methods like, e.g., coverage guided fuzzing. By analyzing the system state space covered by the current model, input sequences which address the gaps in the coverage can be used in the next iterations of the computation. While this increases the required resources, it increases the chance for discovering faults and their causes as well.

  It is possible to use a computed fault seed as such a guidance for future input sequences. If a computed cause is not sufficiently explaining the occurring error, it can help to determine which testbench should be executed next. Consider, e.g., the fault seeds $(WSPEED = 4)$ and $(WSPEED == 2, WSPEED = 4)$ which were computed in Section 3.3.2. The second fault seed was computed after another iteration of executing the wind turbine, for which a different input sequence was used. And, in fact, the first fault seed does imply that more testbenches including $WSPEED$ being set to 4 are necessary.

- **Formal methods:** The only way of ensuring the full system coverage of the extracted model is by applying formal methods such as symbolic execution. By exchanging the concrete input values with symbolic ones, all possible execution paths are covered, which guarantees the completeness of the final model. Then, no faults or their causes can be missed. But the limiting aspect of formal methods is their complexity, which increases the needed resources.

The three proposed types of applying the framework state a pay-off in speed and quality of error detection and explanation. The suitability of each type can depend on the overall quality aims of the system that is being designed or on the current design phase.

4.2 Explanation Quality

As already mentioned, the aim of an explanation in this case is to reduce the produced information to the relevant part. Within the framework, this is achieved by computing the minimal sufficient condition for triggering an error. Further, the importance of both the order and the context of events for causal explanations was made apparent in this paper. The given illustrative example shows that each on their own can be insufficient to properly define the cause of an error. But only specifying the best explanation based on minimal length and by preferring context-independent events compared to context-dependent ones can be misleading:

- Several causes which are rated equally good can be computed, such as the three fault seeds including context-dependent events in Section 3.3.2. By giving too many causes, the relevant information could be hard to detect for a designer. By choosing one cause randomly, the explanation could be unnecessarily complicated or the relevant information could be lost.

- The minimal cause can be too obvious. The algorithm skipped, e.g., the explanation $(WSPEED = 4)$ for Property 1 and the explanation $(PITCH == 0, MODE = 1)$ for Property 2, as they only state **that** the error occurred but yield no further information to **why** this happened. But it is not apparent for every type of property, which explanation would be too obvious and should therefore be excluded.

Hence, further insight into assessing the quality of a causal explanation is necessary. This also applies to the proposed cause-effect chains. While they can add further information by ordering the events of a cause, this additional information doesn't necessarily result in better understanding.

4.3 Scalability & Robustness

While the presented case study serves as a proof-of-concept for the proposed explanation framework, it doesn't suffice to argue about the scalability and robustness of the approach. For this, an application to more use cases, especially more complex, industrial-scale CPSs, is necessary. Then, it can be observed if the proposed computation of causes is sufficient to explain the occurring faults. Further, the complexity of the algorithm for cause computation can be analyzed. It is of interest, how the algorithm can be computed efficiently even for larger model sizes or more complex causes. This could be achieved by, e.g., modifying the algorithm or by further abstracting the model.

4.4 Abstraction Level

The proposed approach aims at explanations at system level. This does not yet exhaust the benefits gained by using VPs. A reasonable future direction is to apply the approach at a lower abstraction level by monitoring internal values of the VP like register values as well. But as this would significantly increase the state space, it is necessary to pursue assessing the current state of scalability and options for improving resource requirements first. Even at system level, the approach is valuable though. The considered use case only uses the functionalities for modeling interaction of HW components and SW while regarding timing. While a VP is not necessary for this, it supports implementing more complex systems in the future and system design, in general, profits by using VPs.

5 Conclusion

In this work, a framework for identifying the cause of a fault at system level was proposed to start on the path toward SXVPs. Based on monitoring the system at runtime by using a VP for RISC-V, a formal model was extracted to be used for computing the fault seed. This fault seed was extended to be context-dependent and to enable cause-effect chains. The framework was demonstrated for an abstracted wind turbine controller, containing a sensor and an actuator.

References

[1] P. Pieper and R. Drechsler, *Formal and Practical Techniques for the Complex System Design Process using Virtual Prototypes*. Springer Cham, 2024.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," in *Vol. 58 of Advances in Computers*, pp. 117–148, 2003.

[3] M. Blumreiter, J. Greenyer, F. J. C. Garcia, V. Klös, M. Schwammberger, C. Sommer, A. Vogelsang, and A. Wortmann, "Towards self-explainable cyber-physical systems," in *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 543–548, 2019.

[4] M. A. Köhl, D. Bohlender, K. Baum, M. Langer, D. Oster, and T. Speith, "Explainability as a non-functional requirement," in *IEEE 27th International Requirements Engineering Conference (RE)*, pp. 363–368, 2019.

[5] G. Fey, M. Fränzle, and R. Drechsler, "Self-explanation in systems of systems," in *Proceedings of the IEEE International Conference on Requirements Engineering*, pp. 85–91, 2022.

[6] D. Minh, H. X. Wang, Y. F. Li, and T. N. Nguyen, "Explainable artificial intelligence: a comprehensive review," in *Artificial Intelligence Review*, pp. 3503—3568, 2021.

[7] S. Jiang, T. E. Fuhrman, and S. K. Jha, "Model checking for fault explanation," in *Proceedings of the IEEE Conference on Decision and Control*, pp. 404–409, 2006.

[8] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*. Springer Cham, 2018.

[9] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, 2020.