A Formalization of Heisenbugs and Their Causes

Sarah Sallinger, Georg Weissenbacher, and Florian Zuleger

TU Wien, Vienna, Austria
firstname.lastname@tuwien.ac.at

Abstract. The already challenging task of identifying the cause of a bug becomes even more cumbersome if those bugs disappear or change their behavior under observation. Such bugs occur in a range of contexts including elusive concurrency bugs as well as unintended system alterations during debugging and—as a pun on the name of Werner Heisenberg—are often referred to as Heisenbugs. Heisenbugs can be caused by various sources of nondeterminism on different system levels, many of which developers and testers might not even be aware of. This paper provides formal foundations for rigorously reasoning about causes of Heisenbugs. It provides a formal definition of Heisenbugs in terms of a hyperproperty and introduces a framework for determining the causality of Heisenbugs in presence of multiple candidate causes based on said hyperproperty. We analyze the properties of causes and the implications on practical causal analyses.

1 Introduction

Bugs which change their behavior under observation are notoriously difficult to detect and fix. Inspired by Heisenberg's uncertainty principle such bugs are often referred to as *Heisenbugs*. Depending on the context, the term Heisenbug has been used to describe slightly different concepts. In the software engineering community, the term is used mostly for bugs whose analysis is hampered by the probe effect, i.e., an unintended alteration of the system behavior during debugging [18]. In the formal methods community, the term has been used to refer to elusive faults arising from executions that exhibit nondeterminism, in particular in the context of concurrent software [30]. In the context of automated testing, the term flaky test is used for inconsistently failing test cases [31], i.e. manifestations of Heisenbugs. As will become apparent in this paper, all the mentioned phenomena can be formalized in a uniform manner. In the rest of the paper, we hence use the term Heisenbug to refer to all the mentioned categories[1].

A Formalization of Heisenbugs. The first contribution of this paper is a formal definition of Heisenbugs. The unifying characteristic of Heisenbugs in the above-mentioned categories is the existence of at least two system executions where one

[1] In the literature, sometimes the term Mandelbug is used as an umbrella term for the mentioned categories. However, Mandelbugs additionally include complex faults where there is "a delay between the fault activation and the final failure occurrence".

execution is correct and the other exhibits a bug. In terms of testing, the same test case sometimes succeeds and sometimes fails. We formalize this definition in terms of a hyperproperty [7], which checks for the existence of two terminating executions with equal inputs but deviating outcomes for a final assertion that is part of the system specification. Our definition accommodates deviations caused by nondeterminism in a single program, e.g. due to concurrency, as well as deviating behavior of different versions of the program, e.g. due to changes for debugging.

Debugging Challenges. Previous studies have shown that Heisenbugs are prevalent even in mature software systems and that the bug fixing process takes significantly longer than for ordinary bugs [9]. Furthermore, Heisenbugs significantly complicate automated testing techniques, as they lead to flaky tests [31].

A major step in the debugging process is the identification of the bug's root causes [35]. Developers reported this step to be particularly difficult for Heisenbugs [11] (referred to as flaky tests in this study). One reason for the complexity is that Heisenbugs can be caused by *mechanisms* (i.e., sources of nondeterminism or system alterations) located on all system levels ranging from the hardware level to the user program. The following examples illustrate some possible causes:

Example 1 (Concurrency). We first present an example for a Heisenbug stemming from system internal nondeterminism. The Therac-25 incident [24,37], which resulted in the death of several cancer patients, is a notorious instance of an atomicity violation [27]. Listing 1.1 illustrates the problem, which is caused by the concurrent execution of two routines: the `userInterface` routine allows the operator to choose between high energy x-ray therapy (`isXray`) and a lower energy electron beam therapy (`!isXray`) and to set the intensity of the radiation (`isHigh`). The assume statement in line 6 prevents a selection of high-intensity electron therapy. The `setup` routine then processes these inputs: a failed assertion represents the case where the patient is exposed to excessive radiation. Assume that the user changes the initial configuration from high-intensity x-ray treatment (`isXray=true`, `isHigh=true`) to low-energy electron therapy (`isXray=false`, `isHigh=false`) in lines 4 and 7. If a context switch occurs right after executing line 5, the assertion in line 13 will fail. When `userInterface` is executed atomically, however, the assertion always holds.

```
1  bool isXray = true;
2  bool isHigh = true;
3  void *userInterface(void *a) {
4    isXray = read();
5    bool isHighTmp = read();
6    assume(isXray || !isHighTmp);
7    isHigh = isHighTmp;
8  }
9  void *setup(void *a) {
10   bool filter = isXray;
11   bool highEnergy = isHigh;
12 }
13 assert(filter || !highEnergy);
```

Listing 1.1: Illustration of Therac-25 atomicity violation

To sum up, there is a Heisenbug caused by different possible schedules, which is an example for the category of Heisenbugs arising from nondeterministic systems.

Even if the scheduler might in fact be deterministic, its internal steps are not observable for the programmer (which we model using nondeterminism).

Example 2 (Floating Point Precision). A prominent example for unintented system alterations are debugging statements that inadvertently change program outcomes. Consider Listing 1.2 (following [28]), which computes the square of 10^{308} and is expected to cause an overflow given the double-precision floating-point representation. When compiled with optimization level `-O3` and executed using x87 instructions, however, the computation results in 10^{308} rather than in an overflow, and the assertion fails. The reason is that the computation uses 80-bit floating point registers and performs rounding only once values are stored in 64-bit memory cells. Adding the `printf` statement in line 4 enforces such a write to memory, thus yielding the expected overflow, and the assertion holds.

```
1 double v = 1E308;
2 double y = 0;
3 y = v * v;
4 // printf("%g\n", y);
5 assert(isinf(y / v));
```

Listing 1.2: Floating-point computation overflows in case of `printf`-debugging

The failing and correct executions actually stem from two different system versions. In the considered execution model, the debugging statement changes the semantics of the program, introducing a probe effect which causes the Heisenbug.

Multiple Causes. While the mechanisms causing the Heisenbugs in Example 1 and Example 2 can still be easily identified, such an analysis becomes more challenging for more complex systems where several such mechanisms interact in a non-trivial manner [33] (as in Example 3 below).

Example 3 (Weak Memory Models). Listing 1.3 shows Peterson's mutual exclusion algorithm for two processes. Computer architectures with weak memory models relax the guarantees on the order in which variable assignments are observed across processor cores, causing the algorithm to fail. In particular, the synchronization fails if both processes set their flags in lines 5 and 15 but do not commit the modifications from cache to shared memory before lines 8 and 18 are executed, thus resulting in a Heisenbug (see [34]). Such a reordering, however, is effectively prevented if there are `printf` statements in lines 7 and 17 (as might be the case during development) and hence the bug only occurs once the `printf` statements are removed. Yet, the `printf` statements are not causally related to the Heisenbug (unlike in Example 2), as we will formally argue in Section 3.

Formal Causality Framework. In order to rigorously determine which mechanisms cause a Heisenbug in settings with multiple candidate causes, we present a formal causality defintion based on Lewis' counterfactuals [26] and the causality framework of Galles and Pearl [14]. In counterfactual reasoning, an event is a cause of an effect, if in an alternative world where the cause does not occur, the effect does also not occur. In a nutshell, in a setting with multiple candidate mechanisms, a subset of the mechanisms is a cause of a Heisenbug if there are

```c
int flagP0 = 0, flagP1 = 0;
int turn = 0;
int critical = 0, error = 0;
void *petersonP0(void *a) {
  flagP0 = 1;
  turn = 1;
  //printf("barrier");
  while (flagP1 && (turn == 1));
  critical++;
  if (critical != 1) error++;
  critical--;
  flagP0 = 0;
}
void *petersonP1(void *a) {
  flagP1 = 1;
  turn = 0;
  //printf("barrier");
  while (flagP0 && (turn == 0));
  critical++;
  if (critical != 1) error++;
  critical--;
  flagP1 = 0;
}
assert(error == 0);
```

Listing 1.3: Peterson's algorithm occasionally fails on weak memory models

correct as well as failing executions which agree on the behavior of all other given mechanisms. This is formalized by means of a hyperproperty resembling our formal definition of Heisenbugs.

Note that our formal definition of causes refers to alternative scenarios for counterfactual reasoning. This requires the sources of nondeterminism to be made explicit in the underlying model (or controllable in the system under test, respectively). In practice, however, identifying and controlling all possible sources of nondeterminism is hardly feasible. Therefore, we prove that our causal analysis yields sound results even if some sources of nondeterminism remain unknown or uncontrollable: the result of evaluating our causality hyperproperty in a nondeterministic system is always a subset of a cause identified in the corresponding determinized system in which all sources are made explicit and controllable.

Based on these results, we present an iterative refinement methodology for causal analysis and discuss practical challenges. We showcase how the methodology can be applied for analyses based on model checking and testing.

Main Contributions. The paper presents:

- A formal definition of Heisenbugs in reactive systems in terms of a hyperproperty, in presence of system-internal nondeterminism and/or unintended system alternations (Section 2).
- A hyperproperty-based approach for defining the causality of Heisenbugs in the presence of several potential causes and nondeterminism (Section 3).
- A methodology for causal analysis based on iterative refinement (Section 4).

2 A Formalization of Heisenbugs

This section provides our system model and a formal definition of Heisenbugs.

2.1 System Model

In the following, the term *formula* refers to a first-order formula with a background theory that fixes the interpretations of predicates and function symbols.

Definition 1. *A* Symbolic Transition System *(STS) is a tuple* $(X, I, \mathsf{init}, \mathsf{final}, T)$*, where* X *and* I *are disjoint sets of* system *and* input variables*, respectively, the* initial condition init *is a formula over* $X \cup I$*, the* final condition final *is a formula over* X*, and the* transition relation T *is a formula over* $X \cup I \cup X'$*, where the variables* X' *denote primed copies of the variables* X*.*

Let Val *be a domain of* values*. Following [38], we assume* Val *to contain a special value* τ *that represents* quiescence*, i.e., the absence of an input. A* configuration c *of an STS is a mapping of the variables in* $(X \cup I)$ *to values in* Val*. A* state s *is a mapping of the variables in* X *to values in* Val*. An* input i *is a mapping of the variables in* I *to values in* Val*. The state* $c|_X$ *resp. input* $c|_I$ *of a configuration* c *is the restriction of the mapping to variables in* X *resp.* I*. We write* $c(v)$ *for the value of a variable* $v \in (X \cup I)$ *in configuration* c *(and we use the same notation for states and inputs).*

For a formula φ *and a mapping* m *of variables to values we write* $m \models \varphi$ *if* φ *evaluates to true under* m*. A configuration* c *is* initial *if* $c \models \mathsf{init}$*. A state* s *is* final *if* $s \models \mathsf{final}$*. A configuration* c *is* final *if* $c|_X$ *is final. A state* $s : X \to \mathsf{Val}$ *is* successor state *of configuration* c *if* $\langle c, s' \rangle \models T$*, where* $s' : X' \to \mathsf{Val}$ *is the function that maps each primed variable* $x' \in X'$ *to* $s(x)$ *and* $\langle \cdot, \cdot \rangle$ *denotes the union of two mappings with disjoint domains. We call* c_{i+1} *a* successor configuration *of* c_i *if* $c_{i+1}|_X$ *is a successor state of* c_i*. We require that final configurations do not have successor configurations.*

A (finite or infinite) trace *of an STS is a sequence of configurations* $c_0, \ldots, c_n$ *where* $c_0 \models \mathsf{init}$*, and* c_{i+1} *is a successor of* c_i *for all* $i \geq 0$*. An* execution *of an STS is a finite trace* $c_0, \ldots, c_n$ *such that* c_n *is a final configuration.*

It is straightforward to represent programs such as the examples from the introduction as symbolic transition systems:

Example 4. Listing 1.1 can be modeled as an STS with $I = \{\mathsf{input}_1, \mathsf{input}_2\}$ and $X = \{\mathsf{isXray}, \mathsf{isHigh}, \mathsf{isHighTmp}, \mathsf{filter}, \mathsf{highEnergy}, \mathsf{pc}_0, \mathsf{pc}_1\}$, where the variables pc_0 and pc_1 model the program counters of the two threads. The initial condition is $(\mathsf{isXray} \wedge \mathsf{isHigh} \wedge \mathsf{pc}_0 = 4 \wedge \mathsf{pc}_1 = 10)$. The final condition is $(\mathsf{pc}_0 = 8 \wedge \mathsf{pc}_1 = 12)$ and describes that both traces have reached their final program location. The transition relation T shown in Figure 1 is a disjunctive partitioning that represents a case split over all possible combinations of program locations, where the thread to be executed in each step is chosen nondeterministically.

While Example 4 illustrates the case of nondeterminism in a single system version, we next exemplify how to model system alterations in our formal model: the original and the altered system can be combined in one STS with an initial nondeterministic choice between two disjuncts of the transition relation.

$$
\begin{array}{llcl}
 & \overbrace{\hphantom{(\mathsf{pc}_1 = 11 \land \mathsf{pc}_1' = 12) \land (\mathsf{pc}_0' = \mathsf{pc}_0)}}^{\text{control flow}} & & \overbrace{\hphantom{(\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{isHigh}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \land (\mathsf{isHigh}' = \mathsf{isHighTmp})}}^{\text{data flow}} \\
 & (\mathsf{pc}_0 = 4 \land \mathsf{pc}_0' = 5) \land (\mathsf{pc}_1' = \mathsf{pc}_1) & \land & (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{isXray}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \land (\mathsf{isXray}' = \mathsf{input}_1) \\
\lor & (\mathsf{pc}_0 = 5 \land \mathsf{pc}_0' = 7) \land (\mathsf{pc}_1' = \mathsf{pc}_1) & \land & (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{isHighTmp}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \land \\
 & & & (\mathsf{isHighTmp}' = \mathsf{input}_2) \land (\mathsf{isXray}' \lor \neg \mathsf{isHighTmp}') \\
\lor & (\mathsf{pc}_0 = 7 \land \mathsf{pc}_0' = 8) \land (\mathsf{pc}_1' = \mathsf{pc}_1) & \land & (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{isHigh}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \land (\mathsf{isHigh}' = \mathsf{isHighTmp}) \\
\lor & (\mathsf{pc}_1 = 10 \land \mathsf{pc}_1' = 11) \land (\mathsf{pc}_0' = \mathsf{pc}_0) & \land & (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{filter}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \land (\mathsf{filter}' = \mathsf{isXray}) \\
\lor & (\mathsf{pc}_1 = 11 \land \mathsf{pc}_1' = 12) \land (\mathsf{pc}_0' = \mathsf{pc}_0) & \land & (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{highEnergy}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \land \\
 & & & (\mathsf{highEnergy}' = \mathsf{isHigh})
\end{array}
$$

Fig. 1: Transition relation for Listing 1.1

Example 5. The floating point program from Listing 1.2 can be modeled as an STS where $X = \{\mathsf{v}, \mathsf{y}, \mathsf{pc}, \mathsf{print}\}$, $I = \emptyset$, the initial condition is $(\mathsf{pc} = 3 \land \mathsf{v} = 10^{308} \land \mathsf{y} = 0)$, the final condition is $\mathsf{pc} = 5$, and the transition relation is defined as $(\mathsf{pc} = 3 \land \mathsf{pc}' = 5 \land \mathsf{y}' = \mathsf{v} * \mathsf{v} \land \mathsf{v}' = \mathsf{v} \land \neg \mathsf{print} \land \mathsf{print}' = \mathsf{print}) \lor (\mathsf{pc} = 3 \land \mathsf{pc}' = 5 \land \mathsf{y}' = \mathsf{convert64}(\mathsf{v} * \mathsf{v}) \land \mathsf{v}' = \mathsf{v} \land \mathsf{print} \land \mathsf{print}' = \mathsf{print})$ where $\mathsf{convert64}$ is a function producing the 64 bit representation of the number. The initial condition does not constrain print, i.e., the initial value of print can be arbitrary; this initial (nondeterministic) choice then fixes the respective disjunct of the transition relation depending on whether the `printf`-statement is present or not.

In the following, we formally define a number of useful properties of STSs.

Definition 2 (Termination). *An STS is* terminating *if the STS does not have infinite traces.*

Definition 3 (Input Determinism). *An STS is* input-deterministic *if 1) for every input i there is at most one state s such that $\langle s, i\rangle \models \mathsf{init}$, and 2) for every state s and every input i there is at most one successor state. Otherwise, it is* nondeterministic*.*

Definition 4 (Input-enabled). *An STS is* input-enabled *if 1) for every input i there is at least one state s such that $\langle s, i\rangle \models \mathsf{init}$, and 2) every configuration that is not final has at least one successor. In case 2) is violated, we call the transition relation* partial*.*

We next define assertions as well as succeeding and failing executions:

Definition 5 (Assertions, Succeeding and Failing Executions). *An* assertion *is a formula φ over the system variables X. An execution $\pi \overset{\text{def}}{=} c_0, \ldots, c_n$* succeeds *with respect to φ if $c_n|_X \models \varphi$. Similarly, π* fails *if $c_n|_X \models \neg\varphi$. Abusing our notation, we write $\pi \models \varphi$ if π succeeds and $\pi \not\models \varphi$ if π fails.*

We note that without input-enabledness, which we do not require in general, traces can get stuck at non-final configurations: For example, in Figure 1, any state with $\mathsf{pc}_0 = 5, \mathsf{pc}_1 = 12, \mathsf{isXray} = \mathsf{false}, \mathsf{input}_2 = \mathsf{true}$ does not have a

control flow		inputs		states				
pc_0	pc_1	input_1	input_2	isXray	isHigh	isHighTmp	filter	highEnergy
4	10	F	τ	T	T	F	F	F
5	10	τ	F	F	T	F	F	F
7	10	τ	τ	F	T	F	F	F
8	10	τ	τ	F	F	F	F	F
8	11	τ	τ	F	F	F	F	F
8	12	τ	τ	F	F	F	F	F

Fig. 2: Execution π_1 of Figure 1

control flow		inputs		states				
pc_0	pc_1	input_1	input_2	isXray	isHigh	isHighTmp	filter	highEnergy
4	10	F	τ	T	T	F	F	F
5	10	τ	F	F	T	F	F	F
7	10	τ	τ	F	T	F	F	F
7	11	τ	τ	F	T	F	F	F
7	12	τ	τ	F	T	F	F	T
8	12	τ	τ	F	F	F	F	T

Fig. 3: Execution π_2 of Figure 1

successor. For such traces, it is not meaningful to argue whether they satisfy an assertion. This is why Definition 5 quantifies over executions, i.e., traces that end in a final configuration. Moreover, Definition 5 disregards infinite traces, as we limit ourselves in this paper to Heisenbugs that are observable in a finite amount of time; we leave the extension to non-terminating traces to future work.

Example 6. Figures 2 and 3 show executions of the STS from Example 4. For $\varphi \overset{\text{def}}{=} (\mathsf{filter} \vee \neg \mathsf{highEnergy})$ (the assertion in line 13), we have $\pi_1 \models \varphi$ and $\pi_2 \not\models \varphi$.

We presume that a system contains a bug if it has at least one failing execution (i.e., we assume that the assertion φ correctly encodes desired behavior of the program):

Definition 6. *Let $(X, I, \mathsf{init}, \mathsf{final}, T)$ be an STS and the assertion φ be a formula over X. The STS contains a* bug *with respect to φ if there exists a failing counterexample execution:*

$$\exists \pi_c \,.\, \pi_c \not\models \varphi.$$

A violation of the property φ in Definition 6, however, does not necessarily constitute a Heisenbug.

2.2 Formal Definition of Heisenbugs

Heisenbugs are special bugs which occur only on some, but not on all executions. We express this in terms of a hyperproperty [7]. Unlike properties over single executions (such as Definition 6), hyperproperties relate sets of traces, allowing us to characterize Heisenbugs by juxtaposing the behavior of two executions. In particular, we require at least one succeeding and one failing execution induced by the same input (as deviating behavior is to be expected for differing inputs). To express this requirement for reactive systems, we define the projection of a trace to its corresponding sequence of inputs that are not quiescent (i.e., not τ):

Definition 7. *Let π be a trace of the STS $(X, I, \mathsf{init}, \mathsf{final}, T)$, and let $J \subseteq I$ be some subset of the input variables. The* input sequence $J(\pi)$ *is defined inductively:*

$$J(\varepsilon) = \varepsilon, \qquad J(c \cdot \pi) = \begin{cases} J(\pi), & \text{if } \forall \mathsf{i} \in J : c(i) = \tau \\ c|_J \cdot J(\pi) & \text{otherwise} \end{cases}$$

where ε is the trace of length zero and $\cdot$ represents concatenation.

Example 7. The traces from Figures 2 and 3 have the same inputs $\langle \mathsf{input}_1 \mapsto \mathsf{false}, \mathsf{input}_2 \mapsto \tau\rangle \cdot \langle \mathsf{input}_1 \mapsto \tau, \mathsf{input}_2 \mapsto \mathsf{false}\rangle$ for $J = I = \{\mathsf{input}_1, \mathsf{input}_2\}$.

Heisenbugs can be characterized using a hyperproperty asserting the existence of two executions with matching inputs, one of which violates the assertion while the other fulfills it:

Definition 8. *An STS $(X, I, \mathsf{init}, \mathsf{final}, T)$ contains a* Heisenbug *with respect to an assertion φ if*

$$\exists \pi_c, \pi_w \,.\, I(\pi_c) = I(\pi_w) \wedge \pi_c \not\models \varphi \wedge \pi_w \models \varphi.$$

The execution π_c is the counterexample execution, *π_w is the* witness execution.

We emphasize that the definition is expressed in terms of a hyperproperty stating that the inputs of the two traces must match. This condition cannot be expressed as a simple trace property. Moreover, we remark that Definition 8 is amenable to hyperproperty model checking (e.g., [13]).

Example 8. The Therac-25 example contains a Heisenbug with counterexample execution π_2 from Figure 3 and witness execution π_1 from Figure 2.

3 Causality

In this section, we extend the hyperproperty from Definition 8 to counterfactually reason about the causality of Heisenbugs. We first present a refinement step for making potential causes explicit in the model and then introduce formal definitions of causality in deterministic as well as nondeterministic systems.

3.1 Modeling Sources of Nondeterminism

For the purpose of causality analysis, the sources of nondeterminism (which we call *mechanisms*) need to be made explicit. Nondeterminism can be due to incomplete observability, incomplete modeling or to inherent stochasticity in the modeled system, as is the case for example in quantum mechanics [15, Section 3.1]. Nondeterminism stemming from incomplete observability and modeling can be eliminated by refining the model with the relevant information. Even true nondeterminism can—at least in principle—be accounted for by means of prophecy variables [1].

To formalize this idea, we introduce refinements of a transition system:

$$\begin{array}{clllc}
 & \overbrace{\phantom{\neg\mathsf{thread}}}^{\text{schedule}} & \overbrace{\phantom{\wedge(\mathsf{pc}_1 = 10 \wedge \mathsf{pc}_1' = 11) \wedge (\mathsf{pc}_0' = \mathsf{pc}_0)}}^{\text{control flow}} & & \overbrace{}^{\text{data flow}} \\
 & \neg\mathsf{thread} & \wedge(\mathsf{pc}_0 = 4 \wedge \mathsf{pc}_0' = 5) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & \dots \\
\vee & \neg\mathsf{thread} & \wedge(\mathsf{pc}_0 = 5 \wedge \mathsf{pc}_0' = 7) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & \dots \\
\vee & \neg\mathsf{thread} & \wedge(\mathsf{pc}_0 = 7 \wedge \mathsf{pc}_0' = 8) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & \dots \\
\vee & \mathsf{thread} & \wedge(\mathsf{pc}_1 = 10 \wedge \mathsf{pc}_1' = 11) \wedge (\mathsf{pc}_0' = \mathsf{pc}_0) & \wedge & \dots \\
\vee & \mathsf{thread} & \wedge(\mathsf{pc}_1 = 11 \wedge \mathsf{pc}_1' = 12) \wedge (\mathsf{pc}_0' = \mathsf{pc}_0) & \wedge & \dots
\end{array}$$

Fig. 4: Deterministic transition relation for Listing 1.1

Definition 9 (Refinement). *Let $S \stackrel{\text{def}}{=} (X, I, \mathsf{init}, \mathsf{final}, T)$ be an STS. We say an STS $S_{\mathsf{ref}} = (X \uplus X_{\mathsf{ref}}, I \uplus I_{\mathsf{ref}}, \mathsf{init}_{\mathsf{ref}}, \mathsf{final}_{\mathsf{ref}}, T_{\mathsf{ref}})$ is a* refinement *of S iff*

1. *for every $\langle\langle s, s_{\mathsf{ref}}\rangle, \langle i, i_{\mathsf{ref}}\rangle, \langle s', s'_{\mathsf{ref}}\rangle\rangle \models T_{\mathsf{ref}}$ we have that $\langle s, i, s'\rangle \models T$, and for every state $\langle s, s_{\mathsf{ref}}\rangle$ of S_{ref} and transition $\langle s, i, s'\rangle \models T$ there are mappings $i_{\mathsf{ref}}, s'_{\mathsf{ref}}$ of I_{ref} and X'_{ref} to values such that $\langle\langle s, s_{\mathsf{ref}}\rangle, \langle i, i_{\mathsf{ref}}\rangle, \langle s', s'_{\mathsf{ref}}\rangle\rangle \models T_{\mathsf{ref}}$,*
2. *for every $\langle\langle s, s_{\mathsf{ref}}\rangle, \langle i, i_{\mathsf{ref}}\rangle\rangle \models \mathsf{init}_{\mathsf{ref}}$ we have $\langle s, i\rangle \models \mathsf{init}$, and for every $\langle s, i\rangle \models \mathsf{init}$ there are mappings $i_{\mathsf{ref}}, s_{\mathsf{ref}}$ of I_{ref} and X_{ref} to values such that $\langle\langle s, s_{\mathsf{ref}}\rangle, \langle i, i_{\mathsf{ref}}\rangle\rangle \models \mathsf{init}_{\mathsf{ref}}$, and*
3. *for every $\langle s, s_{\mathsf{ref}}\rangle \models \mathsf{final}_{\mathsf{ref}}$ we have $s \models \mathsf{final}$, and for every $s \models \mathsf{final}$ and every mapping s_{ref} of the variables X_{ref} to values we have $\langle s, s_{\mathsf{ref}}\rangle \models \mathsf{final}_{\mathsf{ref}}$.*

We note that the above definition preserves executions: Let S_{ref} be a refinement of some STS S. Then every execution of S_{ref} gives rise to an execution of S by projecting away the additional state and input variables. On the other hand, the conditions in the refinement definition ensure that every execution of S can be extended to an execution of S_{ref} by choosing suitable values for the additional state and input variables. We note that refinements can be thought of as adding additional information to the STS under analysis, and the requirements in our definition ensure that executions are preserved. If *all* mechanisms (i.e., sources of nondeterminism) are explicit, refinement yields a deterministic system:

Definition 10 (Determinization). *We say that an STS S_{det} is a* determinization *of some STS S, if S_{det} is a refinement of S and is input-deterministic.*

Example 9. The Therac-25 transition system from Example 4 can be refined by setting $I_{\mathsf{ref}} = \{\mathsf{thread}\}$, where thread is a Boolean variable selecting which thread takes a step. The refined transition relation is shown in Figure 4.

Example 10. The floating point transition system from Example 5 can be extended to a deterministic system by setting $I_{\mathsf{ref}} = \{\mathsf{debug}\}$ and considering the refined initial condition $(\mathsf{pc} = 3 \wedge \mathsf{v} = 10^{308} \wedge \mathsf{y} = 0 \wedge (\mathsf{debug} \Leftrightarrow \mathsf{print}))$, and leaving the transition relation unchanged. We point out that the initial value of the input variable debug fixes the value of print, which in turn fixes the transition relation reflecting the presence of the `printf`-statement.

For Example 9 and Example 10 we have $X_{\mathsf{ref}} = \emptyset$. In the Peterson example below, the refinement contains a state variable reflecting whether the cache state has been propagated to main memory.

$$
\begin{array}{lll}
 & \overbrace{\hphantom{((\mathsf{pc}_0 = 5 \wedge \mathsf{pc}'_0 = 6) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1))}}^{\text{control flow}} & \overbrace{\hphantom{(\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{flagP0c}, \mathsf{flagP0}\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{flagP0c}' = 1) \wedge}}^{\text{data flow}} \\
 & ((\mathsf{pc}_0 = 5 \wedge \mathsf{pc}'_0 = 6) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{flagP0c}, \mathsf{flagP0}\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{flagP0c}' = 1) \wedge \\
 & & (\mathsf{delay} \wedge \mathsf{flagP0}' = \mathsf{flagP0} \vee \neg\mathsf{delay} \wedge \mathsf{flagP0}' = 1) \\
\vee & (\mathsf{pc}_0 = 6 \wedge \mathsf{pc}'_0 = 7) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{turn}\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{turn}' = 1) \\
\vee & (\mathsf{pc}_0 = 7 \wedge \mathsf{pc}'_0 = 8) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in V} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{print} \Rightarrow \neg\mathsf{delay}) \\
\vee & (\mathsf{pc}_0 = 8 \wedge \mathsf{pc}'_0 = 9) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in V} \mathsf{var}' = \mathsf{var} \wedge (\neg\mathsf{flagP1} \vee \mathsf{turn} = 0) \\
\vee & (\mathsf{pc}_0 = 9 \wedge \mathsf{pc}'_0 = 10) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{critical}, \mathsf{flagP0}\}} \mathsf{var}' = \mathsf{var}) \wedge \\
 & & (\mathsf{critical}' = \mathsf{critical} + 1) \wedge \\
 & & (\neg\mathsf{delay} \wedge \mathsf{flagP0}' = \mathsf{flagP0} \vee \mathsf{delay} \wedge \mathsf{flagP0}' = \mathsf{flagP0c}) \\
\vee & (\mathsf{pc}_0 = 10 \wedge \mathsf{pc}'_0 = 11) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{error}\}} \mathsf{var}' = \mathsf{var}) \wedge \\
 & & (\mathsf{critical} \neq 1 \Rightarrow \mathsf{error}' = \mathsf{error} + 1) \wedge \\
 & & (\mathsf{critical} = 1 \Rightarrow \mathsf{error}' = \mathsf{error}) \\
\vee & (\mathsf{pc}_0 = 11 \wedge \mathsf{pc}'_0 = 12) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{critical}\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{critical}' = \mathsf{critical} - 1) \\
\vee & (\mathsf{pc}_0 = 12 \wedge \mathsf{pc}'_0 = 13) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{flagP0c}, \mathsf{flagP0}\}} \mathsf{var}' = \mathsf{var}) \wedge \\
 & & (\mathsf{flagP0c}' = 0) \wedge (\mathsf{flagP0}' = 0))
\end{array}
$$

Fig. 5: A part of T_{ref} for Listing 1.3 (where $V \overset{\text{def}}{=} (X \cup X_{\mathsf{ref}}) \setminus \{\mathsf{pc}_0, \mathsf{pc}_1\}$)

Example 11. Peterson's algorithm (Listing 1.3) can be modeled as a determinisitic STS. In this example we present the final refinement that makes all involved mechnisms explicit. Alternatively, the mechanisms could be made explicit in successive refinement steps. Figure 5 shows the part of the transition relation that models $\mathsf{P0}$. Let $X = \{\mathsf{pc}_0, \mathsf{pc}_1, \mathsf{flagP0}, \mathsf{flagP0c}, \mathsf{flagP1}, \mathsf{flagP1c}, \mathsf{turn}, \mathsf{critical}, \mathsf{error}, \mathsf{print}\}$ where $\mathsf{flagP0c}$ and $\mathsf{flagP1c}$ represent the locally cached versions of the flags. We have $I = \emptyset$ and $I_{\mathsf{ref}} = \{\mathsf{thread}, \mathsf{debug}, \mathsf{reorder}\}$, where thread indicates whether $\mathsf{P0}$ or $\mathsf{P1}$ takes a step (thread is omitted in Figure 5). Let $X_{\mathsf{ref}} = \{\mathsf{delay}\}$, and let $\mathsf{init}_{\mathsf{ref}}$ imply ($\mathsf{print} = \mathsf{debug} \wedge \mathsf{delay} = \mathsf{reorder}$). The variable print indicates that the program version with `printf`-debugging is executed, and delay is true if the modifications of the flags $\mathsf{flagP0}$ and $\mathsf{flagP1}$ are only committed to shared memory after entering the critical section (to avoid clutter, we assume only two possible points for committing the modification of $\mathsf{flagP0}$). We use ($\mathsf{print} \Rightarrow \neg\mathsf{delay}$) to model the interplay between two mechanisms where the `printf` instruction prevents reordering because of the added barrier, resulting in a partial transition relation. Moreover, $\mathsf{init}_{\mathsf{ref}}$ ensures that $\mathsf{flagP0} = \mathsf{flagP0c} = \mathsf{flagP1} = \mathsf{flagP1c} = \mathsf{turn} = \mathsf{critical} = \mathsf{error} = 0$ and $\mathsf{pc}_0 = 5$ and $\mathsf{pc}_1 = 15$, and $\mathsf{final}_{\mathsf{ref}}$ is $\mathsf{pc}_0 = 13 \wedge \mathsf{pc}_1 = 23$.

Note that the processor running the original nondeterministic version of Peterson's algorithm already has micro-architectural features that facilitate instruction reordering (not modeled in Example 11); the auxiliary input $\mathsf{reorder}$ and variable delay merely make this mechanism observable.

3.2 Defining Causes

In the following, we provide a formal definition of causes inspired by Lewis' counterfactuals [26] and the causality framework of Galles and Pearl [14].

Definition 11 (Cause). *Let $S \overset{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$ be a deterministic STS, where I and M are disjoint sets of inputs, and let φ be an assertion. Let $M = M_C \uplus M_N$. We say that M_C is a* cause *with respect to M and φ and iff*

$$\exists \pi_c, \pi_w \,.\, I(\pi_c) = I(\pi_w) \wedge M_N(\pi_c) = M_N(\pi_w) \wedge \pi_c \not\models \varphi \wedge \pi_w \models \varphi \qquad (1)$$

and M_C is a minimal subset of M with this property.

We note that in the above definition we require the inputs I to agree on the executions π_c and π_w, while only the inputs M may differ. The rationale is that we want to apply this definition for studying the causes of Heisenbugs: We are given some (nondeterministic) STS with inputs I, which has a Heisenbug. We now consider some determinization of the STS to which we have added inputs M, modelling the *mechanisms* responsible for the nondeterminism. The above definition then allows to study the cause among the modelled mechanisms: A subset $M_C \subseteq M$ is a cause of a Heisenbug, if the Heisenbug still occurs when the inputs M_N agree in the deviating executions π_c and π_w.

Proposition 1 (Existence of a Cause). *Let $S \overset{\text{def}}{=} (X, I, \mathsf{init}, \mathsf{final}, T)$ be a nondeterministic STS with a Heisenbug (Definition 8) with respect to an assertion φ and let $S_{\mathsf{det}} \overset{\text{def}}{=} (X \cup X_{\mathsf{det}}, I \cup M, \mathsf{init}_{\mathsf{det}}, \mathsf{final}_{\mathsf{det}}, T_{\mathsf{det}})$ be a determinization of S. Then there exists a cause M_C with respect to M and φ.*

Proof. Let π_c and π_w be executions of S that satisfy Definition 8. Since refinements preserve executions, there must be executions $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ of S_{det} such that $\pi_{c\mathsf{det}}|_{(I \cup X)} = \pi_c$ and $\pi_{w\mathsf{det}}|_{(I \cup X)} = \pi_w$. Now assume that $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ agree on M (in addition to I). Let $\langle s_c, s_{c\mathsf{det}} \rangle$ and $\langle s_w, s_{w\mathsf{det}} \rangle$ be the initial states of $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$, respectively. Since S_{det} is input-deterministic, however, there is at most one state $\langle \langle s, s_{\mathsf{det}} \rangle, \langle i, m \rangle \rangle \models \mathsf{init}_{\mathsf{det}}$, hence $\langle s_c, s_{c\mathsf{det}} \rangle = \langle s_w, s_{w\mathsf{det}} \rangle$. Moreover, for every state $\langle s, s_{\mathsf{det}} \rangle$, each input $\langle i, m \rangle$ determines a unique successor state $\langle s', s'_{\mathsf{det}} \rangle$. Since $\pi_{c\mathsf{det}}|_{(I \cup M)} = \pi_{w\mathsf{det}}|_{(I \cup M)}$, this violates the assumption that $\pi_{c\mathsf{det}} \not\models \varphi$ and $\pi_{w\mathsf{det}} \models \varphi$. Hence, $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ must deviate on M. $\square$

Example 12. The Peterson example contains a Heisenbug with respect to $\varphi \overset{\text{def}}{=} (\mathsf{error} = 0)$. Here, $\{\mathsf{reorder}\}$ and $\{\mathsf{thread}\}$ are causes, but $\{\mathsf{debug}\}$ is not: The set $\{\mathsf{reorder}\}$ is a cause because of two executions which both have $\mathsf{debug} = \mathsf{false}$ and the same schedule interleaving the critical sections, but only one execution sets $\mathsf{reorder} = \mathsf{true}$ and hence exhibits the bug. The set $\{\mathsf{thread}\}$ is a cause because of two executions which both have $\mathsf{debug} = \mathsf{false}$ and $\mathsf{reorder} = \mathsf{true}$ where one execution uses a sequential schedule of the two processes and the second execution uses a schedule interleaving the critical sections. Only the second execution exhibits the bug. However, the set $\{\mathsf{debug}\}$ is not a cause because any two executions would either both have to set $\mathsf{reorder} = \mathsf{false}$, making the bug impossible or both set $\mathsf{reorder} = \mathsf{true}$. In this case, by counterposition the constraint $(\mathsf{print} \Rightarrow \neg\mathsf{delay})$ enforces $\mathsf{debug} = \mathsf{false}$, yielding a bug on both executions if the schedule interleaves the critical sections or on no execution otherwise.

3.3 Causes and Nondeterminism

By introducing the notion of a *contributing cause* below, we show that even in the presence of nondeterminism we can still provide guarantees.

Definition 12 (Contributing Cause). *Let $S \overset{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$ be a (potentially nondeterministic) STS, where I and M are disjoints set of inputs, and let φ, $M = M_C \uplus M_N$ satisfy the conditions in Definition 11. We call M_C a* contributing cause *of a Heisenbug.*

We argue that any contributing cause must be a subset of a cause in a corresponding determinization:

Theorem 1. *Let $S \overset{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$ be a nondeterministic STS and let $S_{\mathsf{det}} \overset{\text{def}}{=} (X \cup X_{\mathsf{det}}, I \cup M \cup J, \mathsf{init}_{\mathsf{det}}, \mathsf{final}_{\mathsf{det}}, T_{\mathsf{det}})$ be a determinization of S. Let M_C be a contributing cause in S with respect to M and assertion φ. Then, there exists a cause C in S_{det} with respect to $M \cup J$ and φ such that $M_C \subseteq C \setminus J$.*

Proof. Consider two executions π_c and π_w satisfying Definition 12 for S. Since refinement preserves executions, there must be executions $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ in S_{det} such that $\pi_{c\mathsf{det}}|_{(I \cup M \cup X)} = \pi_c$ and $\pi_{w\mathsf{det}}|_{(I \cup M \cup X)} = \pi_w$ and $\pi_{c\mathsf{det}} \not\models \varphi$ and $\pi_{w\mathsf{det}} \models \varphi$. By Definition 12, for $M_N = M \setminus M_C$ it holds that $\pi_c|_{(I \cup M_N)} = \pi_w|_{(I \cup M_N)}$ and hence also $\pi_{c\mathsf{det}}|_{(I \cup M_N)} = \pi_{w\mathsf{det}}|_{(I \cup M_N)}$. Hence (following an argument similar to the one for Proposition 1) we argue that $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ must deviate on a subset of $M_C \cup J$, i.e., there exists a cause C satisfying Definition 11 such that $C \subseteq M_C \cup J$. Now assume that $M_C \not\subseteq C$. Then M_C is not minimal, since $(M_C \cap C)$ also constitutes a contributing cause. Thus, we must have $M_C \subseteq C \setminus J$. □

Example 13. The refined STS in Example 9 is nondeterministic as the initial values of filter and $\mathsf{highEnergy}$ are unconstrained. Following Definition 12, $\{\mathsf{thread}\}$ is a contributing cause. Consider a further refinement with $I_{\mathsf{ref}} = \{\mathsf{initF}, \mathsf{initH}\}$ and $\mathsf{init} = (\mathsf{filter} = \mathsf{initF} \wedge \mathsf{highEnergy} = \mathsf{initH} \wedge \mathsf{isXray} \wedge \mathsf{isHigh} \wedge \mathsf{pc}_0 = 4 \wedge \mathsf{pc}_1 = 10)$. As the initial values are never read, the cause is again $\{\mathsf{thread}\}$.

We provide a condition under which contributing causes are also causes:

Definition 13 (Cause in Presence of Nondeterminism). *Consider a (potentially nondeterministic) STS $S \overset{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$ such that for all traces π, π' of S with $\pi|_{I \cup M} = \pi'|_{I \cup M}$ we have that*

1. *π ends in a final state if and only if π' ends in a final state,*
2. *$\pi \models \varphi$ if and only if $\pi' \models \varphi$ (in case both traces end in a final state).*

Let φ, $M = M_C \uplus M_N$ satisfy the conditions in Definition 11. We say that M_C is a cause in presence of nondeterminism *with respect to M and φ.*

We will next state a justification for the introduction of the above definition. We first establish that input-enabled determinizations always exist:

Proposition 2. *Let $S \overset{\text{def}}{=} (X, I, \mathsf{init}, \mathsf{final}, T)$ be an input-enabled STS. Then, a deterministic input-enabled refinement S_{ref} always exists.*

Proof. We set $I_{\mathsf{ref}} = \{\mathsf{oracle}\}$ for a single variable oracle, whose values are mappings of configurations to successors, i.e., oracle fixes a successor state s' for every configuration $\langle s, i\rangle$ such that $\langle s, i, s'\rangle \models T$ (note that at least one successor state s' always exists because of our assumption that S is input-enabled). We then adopt T_{ref} from T as the transition relation that moves to the successor state fixed by the oracle variable. Likewise, we adopt the initial condition $\mathsf{init}_{\mathsf{ref}}$. □

We next establish that no matter the input-enabled determinization S' of an STS S, a cause in the presence of nondeterminism in S is always a cause in S'. Together with Proposition 2, which guarantees the existence of an input-enabled determinization at least in theory, we obtain that a cause in presence of nondeterminism can indeed by considered as a cause.

Theorem 2. *Let M_C be a cause in presence of nondeterminism with respect to mechanisms M in an STS $S \overset{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$. Let $S_{\mathsf{det}} \overset{\text{def}}{=} (X \cup X_{\mathsf{det}}, I \cup M \cup J, \mathsf{init}_{\mathsf{det}}, \mathsf{final}_{\mathsf{det}}, T_{\mathsf{det}})$ be an input-enabled determinization of S. Then M_C is also a cause in S_{det} with respect to $(I \cup J)$.*

Proof. Let π_c and π_w be executions of S that satisfy Definition 13. Since refinements preserve executions, there must be an execution $\pi_{c\mathsf{det}}$ of S such that $\pi_{c\mathsf{det}}|_{(I\cup M\cup X)} = \pi_c$. In particular, we have $\pi_{c\mathsf{det}} \not\models \varphi$. Because S_{det} is input-enabled we can obtain a trace π of S_{det} such that $\pi|_J = \pi_{c\mathsf{det}}|_J$ and $\pi|_{I\cup M} = \pi_w|_{I\cup M}$. Note that π induces a trace π' of S with $\pi' = \pi_{w\mathsf{det}}|_{I\cup M\cup X}$. Hence, by the assumptions stated in Definition 13, the trace π' is in fact an execution (i.e., ends with a final configuration), and we have $\pi' \models \varphi$. Thus, we also get that π is an execution and that we have $\pi \models \varphi$. □

Example 14. The nondeterministic refinement of the Therac-25 STS in Example 9 satisfies the properties in Definition 13. The refinement in Example 13 is input-enabled and deterministic and the contributing cause is indeed a cause.

3.4 Testing and Causal Analysis

In the context of testing, an evaluation of Definition 11 and Definition 12, respectively, is limited to the subset of the executions induced by a given test suite. Lemma 1 characterizes the results that can be drawn by analyzing a subset of the executions of an STS:

Lemma 1. *Let π_c and π_w be executions satisfying Equation 1 in Definition 11 (or Definition 12, respectively) and let M_C be the inputs deviating in π_c and π_w. Then M_C is a superset of a cause (or contributing cause, respectively).*

Proof. Note that M_C is a cause according to Definition 11 (or a contributing cause according to Definition 12) if it is minimal with respect to Equation 1. Otherwise, there must be a cause that is a subset of M_C. □

Lemma 1 provides guarantees even if an exhaustive analysis is infeasible. If, in addition, the conditions in Definition 13 are met (i.e., we can control or at least observe the relevant mechanisms), then Proposition 2, Theorem 2, and Lemma 1 guarantee that each overapproximation of a cause identified by testing includes a *non-empty* (contributing) cause.

4 Analysis Methodology and Challenges

We sketch an (iterative) methodology for practical analyses based on the formalization above and showcase two possible instantiations and their challenges:

① **Task:** Starting from a Heisenbug (Definition 8), identify candidate mechanisms M (e.g., consulting surveys [31]).
Challenge: The accuracy of the analysis is contingent on identifying the relevant mechanisms.

② **Task:** Pick a mechanism $\mathsf{m} \in M$ and adapt (or refine according to Definition 9) the model or system to make m controllable (or at least observable).
Challenge: The system may be inherently uncontrollable or unobservable, or attempts to control/observe it potentially introduce a probe effect.

③ **Task:** Identify (contributing) causes by finding witnesses that deviate in as few mechanisms as possible (i.e., satisfy Equation 1 in Definition 11).
Challenge: Testing will yield over-approximations only (cf. Lemma 1).

④ **Task:** Check a stopping criterion to determine whether further mechanisms or refinement steps are required (steps ① and ②).
Challenge: Assessing whether all causes have been correctly identified is challenging and may amount to fixing the bug and re-verifying the system.

Causal Analysis based on Model Checking. We built a NuSMV [5] model of Peterson's algorithm (Listing 1.3). We use self-composition [3], which composes two copies S_w and S_c of the STS S, to reduce the existence of a counterexample trace and a witness trace (which is a hyperproperty) to the existence of a single trace in the composed model. NuSMV can then construct the trace as a counterexample to an LTL property over the composed model. As NuSMV usually considers infinite traces, final conditions are accounted for in the property. The existence of a Heisenbug can be confirmed by checking that NuSMV finds a counterexample to the property $\psi := G(\mathsf{final}_c \wedge \mathsf{final}_w \Rightarrow (\varphi_w \Rightarrow \varphi_c))$ for final and φ as in Example 11 and Example 12 (where subscripted predicates range over the matching variable set).

In step ①, we pick the fact whether the print statements are executed and model it adding variables print_w and print_c to the model (step ②). In step ③, we invoke NuSMV on the property $G(\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \Rightarrow \psi$. As there is a counterexample, we identify the empty set as a contributing cause.

We start another refinement iteration, pick concurrency as machanism (step ①) and model it by variables thread_w and thread_c (step ②). We check the property $G((\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \wedge (\mathsf{thread}_w \Leftrightarrow \mathsf{thread}_c)) \Rightarrow \psi$ (step ③). Again, there is a counterexample and the empty set is a contributing cause.

```
 1 bool flag0 = false;
 2 bool flag1 = false;
 3 spinlock_t lock0, lock1;
 4 void *thread0(void*) {
 5   spin_lock(lock0);
 6     flag0 = true;
 7     assert (!flag1);
 8     yield();
 9     spin_lock(lock1);
10       flag0 = false;
11     spin_unlock(lock1);
12     yield();
13   spin_unlock(lock0);
14 }
15 void *thread1(void*) {
16   spin_lock(lock1);
17     flag1 = true;
18     assert (!flag0);
19     yield();
20     spin_lock(lock0);
21       flag1 = false;
22     spin_unlock(lock0);
23     yield();
24   spin_unlock(lock1);
25 }
```

Listing 1.4: An assertion fails if (and only if) a deadlock occurs.

In the next refinement iteration, we pick the weak memory behavior (step ①) we model it by variables delay_w and delay_c and reflect the fact that $\mathsf{print} \implies \neg\mathsf{delay}$ (cf. Example 11) (step ②). Checking property $G((\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \wedge (\mathsf{thread}_w \Leftrightarrow \mathsf{thread}_c) \wedge (\mathsf{delay}_w \Leftrightarrow \mathsf{delay}_c)) \Rightarrow \psi$ returns true, hence we have now found a non-empty cause superset and can start cause minimization. A counterexample to $G((\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \wedge (\mathsf{thread}_w \Leftrightarrow \mathsf{thread}_c)) \Rightarrow \psi$ witnesses that delay is a cause, similarly a counterexample to $G((\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \wedge (\mathsf{delay}_w \Leftrightarrow \mathsf{delay}_c)) \Rightarrow \psi$ witnesses that thread is a cause. As the model satisfies $G((\mathsf{delay}_w \Leftrightarrow \mathsf{delay}_c) \wedge (\mathsf{thread}_w \Leftrightarrow \mathsf{thread}_c)) \Rightarrow \psi$, print is not a cause. This concludes step ③. As we identified a non-empty cause, no more refinement steps are needed.

Test-based Causal Analysis. Consider the code in Listing 1.4, which might deadlock because of a faulty locking discipline. The assertions in lines 7 and 18 fail when a deadlock, caused by a specific (combination of) context switche(s), occurs: a context switch at line 8 to `thread1` (or, symmetrically, from line 19 to `thread0`) causes both threads to wait for a lock held by the other thread.

In step ①, we identify concurrency (limited to the context switches marked by `yield` for simplicity) as potential cause. Following the approach of KISS [32], we control the scheduler (step ②) by sequentializing the concurrent program and simulating the execution of a large subset of its interleavings. In KISS, threads can be started and terminated nondeterministically at any point during the execution. Using closures to save the local state of a thread, we add the capability to *re-enter* a thread after its interruption by `yield`. The execution of `thread0` (`thread1`, respectively) can be interrupted at lines 8 and 12 (19 and 23, respectively). Our sequentialization enables us to explicitly control these four context switches, inducing 2^4 potential schedules. Random (or systematic) exploration of these schedules then yields executions that terminate normally or violate an assertion. Failing executions deviate from the non-failing ones by performing a context switch at lines 8 or 19, at least one of which must constitute (part of) the candidate cause(s) we identify in step ③.

Testing merely provides an over-approximation of the cause M_C (Lemma 1). Due to the minimality requirement in Definition 11 and Definition 12, however, removing one element from M_C (by controlling the mechanism accordingly) eliminates the entire cause. Assume for now, that `thread0` in Listing 1.4 always executes first, in which case the context switch at line 8 is a unique cause for the deadlock. Consider an over-approximation comprising of two context switches at lines 8 and 19. Blocking the context switch at line 8 eliminates the Heisenbug, while blocking the one at line 19 doesn't. By individually blocking the context switches and checking whether subsequent testing provides sufficient confidence that the bug has been eliminated, we obtain a stopping criterion in step ④.

If, however, executions may start with `thread0` or `thread1`, the context switches at lines 8 and 19 form two independent (non-intersecting) causes (due to the symmetry in Listing 1.4). Consequently, *both* context switches must be identified to eliminate all causes of the bug (cf. Section 3.4). Blocking individual context switches (as suggested above) does not provide a reliable stopping criterion. Despite this limitation, testing-based analysis can help the developer to narrow down the set of candidate causes significantly.

5 Related Work

Terminology and Definition of Heisenbugs. The first paper mentioning Heisenbugs [17] uses the term for transient software bugs which disappear under observation. In [18], bugs are classified into Bohrbugs (bugs manifesting consistently), Mandelbugs (bugs with complex error propagation), and Heisenbugs (bugs manifesting differently under the probe effect). In contrast to this informal classification, our definition is formal, covering Heisenbugs which stem from the probe effect as well as from nondeterminism. The term is frequently (and informally) used in the context of concurrency [30], where it exclusively refers to bugs caused by control-flow nondeterminism. In the context of testing, the notion of flaky tests [31] resembles the notion of Heisenbugs. The comparison of failing and non-failing executions is used in several lines of research with goals orthogonal to the definition of bug classes. Differential assertion checking [21] compares failing and non-failing executions to define relative correctness of different program versions. In the context of diagnosability, the notion of critical pairs of failing and non-failing executions with equivalent observations is used to check whether faults can be detected at runtime [6].

Causality. Our definition of causality is inspired by Lewis' counterfactuals [25]. The negation of Definition 11 mirrors the definition of causal irrelevance in [14] and Definition 11 corresponds to its dual notion of causality between variables [12]. A core difference is that our interventions are restricted to inputs that represent nondeterministic mechanisms rather than affecting arbitrary points of the transition relation (or the causal model). Moreover, causal models have a fixed propagation depth, while we consider an arbitrary number of unwindings of the transition relation. Halpern and Pearl [20,19] provide a widely accepted definition of "actual" causes based on counterfactuals, where contingencies are used to control interference between interventions. Several lines of work reason about the origin of system faults [23,2,10,4,16] using Halpern and Pearl's notion of causality. In [8], actual causality is used to explain violations of hyperproperties. It formalizes causes for violations of (arbitrary) universally quantified hyperproperties as a hyperproperty with quantifier alternation, which can then be checked with a model checker such as [13]. We formalize causes for Heisenbugs (a specific hyperproperty) in terms of an existentially quantified hyperproperty.

Several approaches exist for automatically detecting causes of flaky tests. The RootFinder tool [22] collects passing and failing executions and correlates their differences with a specific cause. In [39] the authors present a tool for finding code locations that lead to differences between succeeding and failing executions. Identifying what happens in these locations is left to the developer. In [36] and [29] the system is repeatedly executed under different configurations to check which configuration influences the manifestation of the bug. All of these approaches are based on computing correlations rather than performing rigorous causal inference. In contrast, our framework is based on a formal causal analysis accounting for interactions of multiple potential causes. [31] provides a taxonomy of causes relevant in the context of automated testing.

6 Conclusion

While the term Heisenbug is widely used, its exact meaning often depends on the context. We provide a formal definition that unifies the notion of Heisenbugs caused by a system alteration and those caused by nondeterminism. Furthermore, we present a hyperproperty-based framework for determining which mechanisms cause the manifestation of a Heisenbug. In particular, our approach allows the identification of causes in the presence of multiple mechanisms that could trigger a Heisenbug and gives guarantees for results of a causal analysis even in presence of nondeterminism. Building on this result, we sketch a methodology for causal analysis based on iterative refinement.

Acknowledgements This work was partially supported by ERC CoG ARTIST 101002685, by the FWF project W1255-N23, by a netidee scholarship, and by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG11005].

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. In: LICS (1988)
2. Baier, C., Dubslaff, C., Funke, F., Jantsch, S., Majumdar, R., Piribauer, J., Ziemek, R.: From verification to causality-based explications. In: ICALP (2021)
3. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM (2011)
4. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: CAV (2009)
5. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: CAV (2002)
6. Cimatti, A., Pecheur, C., Cavada, R.: Formal verification of diagnosability via symbolic model checking. In: IJCAI (2003)
7. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6) (2010)
8. Coenen, N., Dachselt, R., Finkbeiner, B., Frenkel, H., Hahn, C., Horak, T., Metzger, N., Siber, J.: Explaining hyperproperty violations. In: CAV (2022)
9. Cotroneo, D., Grottke, M., Natella, R., Pietrantuono, R., Trivedi, K.S.: Fault triggers in open-source software: An experience report. In: ISSRE (2013)
10. Dubslaff, C., Weis, K., Baier, C., Apel, S.: Causality in configurable software systems. In: ICSE (2022)
11. Eck, M., Palomba, F., Castelluccio, M., Bacchelli, A.: Understanding flaky tests: The developer's perspective. In: ESEC/FSE (2019)
12. Eiter, T., Lukasiewicz, T.: Complexity results for structure-based causality. Artif. Intell. **142**(1) (2002)
13. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: CAV (2015)
14. Galles, D., Pearl, J.: Axioms of causal relevance. Artif. Intell. **97**(1-2) (1997)
15. Goodfellow, I.J., Bengio, Y., Courville, A.C.: Deep Learning. Adaptive computation and machine learning, MIT Press (2016)
16. Gössler, G., Stefani, J.B.: Causality analysis and fault ascription in component-based systems. Theor. Comput. Sci. **837** (2020)
17. Gray, J.: Why do computers stop and what can be done about it? Tech. Rep. 85.7, PN87614, Tandem Computers (June 1986)
18. Grottke, M., Trivedi, K.S.: A classification of software faults. In: ISSRE (2005)
19. Halpern, J.Y.: A modification of the halpern-pearl definition of causality. In: IJCAI (2015)
20. Halpern, J.Y., Pearl, J.: Causes and explanations: A structural-model approach: Part 1: Causes. British Journal for the Philosophy of Science **56** (2005)
21. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: ESEC/FSE (2013)
22. Lam, W., Godefroid, P., Nath, S., Santhiar, A., Thummalapenta, S.: Root causing flaky tests in a large-scale industrial setting. In: ISSTA (2019)
23. Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. In: VMCAI (2013)
24. Leveson, N., Turner, C.: An investigation of the Therac-25 accidents. IEEE Computer **26**(7) (1993)
25. Lewis, D.: Causation. Journal of Philosophy **70**(17) (1974)
26. Lewis, D.: Counterfactuals. Wiley-Blackwell (2001)

27. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access-interleaving invariants. IEEE Micro **27**(1) (2007)
28. Monniaux, D.: The pitfalls of verifying floating-point computations. TOPLAS **30**(3) (2008)
29. Moran, J., Augusto Alonso, C., Bertolino, A., de la Riva, C., Tuya, J.: Flakyloc: Flakiness localization for reliable test suites in web applications. JWE (2020)
30. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI (2008)
31. Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P.: A survey of flaky tests. TOSEM **31**(1) (2021)
32. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI (2004)
33. Ratliff, Z.B., Kuhn, D.R., Kacker, R.N., Lei, Y., Trivedi, K.S.: The relationship between software bug type and number of factors involved in failures. In: ISSRE Wksp (2016)
34. Senftleben, M.: Operational Characterization of Weak Memory Consistency Models. Master's thesis, University of Kaiserslautern (2013)
35. Sommerville, I.: Software Engineering. Addison-Wesley, 9 edn. (2010)
36. Terragni, V., Salza, P., Ferrucci, F.: A container-based infrastructure for fuzzy-driven root causing of flaky tests. In: ICSE (2020)
37. Thomas, M.: The story of the therac-25 in lotos. High Integrity Systems Journal (1994)
38. Tretmans, J.: Test generation with inputs, outputs, and quiescence. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (1996)
39. Ziftci, C., Cavalcanti, D.: De-Flake your tests : Automatically locating root causes of flaky tests in code at google. In: ICSME (2020)