Merit and Blame Assignment with Kind 2[⋆]

Daniel Larraz[1], Mickaël Laurent[12], and Cesare Tinelli[1]

[1] Department of Computer Science, The University of Iowa. USA
[2] IRIF, CNRS — Université de Paris, France

Abstract. We introduce two new major features of the open-source model checker Kind 2 which provide traceability information between specification and design elements such as assumptions, guarantees, or other behavioral constraints in synchronous reactive system models. This new version of Kind 2 can identify minimal sets of design elements, known as *Minimal Inductive Validity Cores*, which are sufficient to prove a given set of safety properties, and also determine the set of *MUST* elements, design elements that are necessary to prove the given properties. In addition, Kind 2 is able to find minimal sets of design constraints, known as *Minimal Cut Sets*, whose violation leads the system to an unsafe state. The computed information can be used for several purposes, including assessing the quality of a system specification, tracking the safety impact of model changes, and analyzing the tolerance and resilience of a system against faults or cyber-attacks. We describe these new capabilities in some detail and report on an initial experimental evaluation of some of them.

Keywords: SMT-based Model Checking · Inductive Validity Cores · Traceability · MUST-set Generation · Minimal Cut Sets · Max-SMT

1 Introduction

Kind 2 [7] is an SMT-based model checker for safety properties of finite- and infinite-state synchronous reactive systems. It takes as input models written in an extension of the Lustre language [13] that allows the specification of assume-guarantee-style contracts for system components. Kind 2's contract language [6] is expressive enough to allow one to represent any (LTL) regular safety property by recasting it in terms of invariant properties. One of Kind 2's distinguishing features is its support for modular and compositional analysis of hierarchical and multi-component systems. Kind 2 traverses the subsystem hierarchy bottom-up, analyzing each system component, and performing fine-grained abstraction and refinement of the sub-components. At the component level, Kind 2 runs concurrently several model checking engines which cooperate to prove or disprove contracts and properties. In particular, it combines two induction-based model checking techniques, k-induction [17] and IC3 [5], with various auxiliary invariant generation methods.

One clear strength of model checkers is their ability to return precise error traces witnessing the violation of a given safety property. In addition to being invaluable to help identify and correct bugs, error traces also represent a checkable unsafety certificate. Similarly, many model checkers are currently able to return some form of corroborating evidence when they declare a safety property to be satisfied by a system under analysis. For instance, Kind 2 can produce an independently checkable proof certificate for the properties that it claims to have proven [15]. However, these certificates, in the form of a k-inductive invariant, give limited user-level insight on what elements of the system model contribute to the satisfaction of the properties.

Contributions We describe two new features of Kind 2 that provide more insights on verified properties: (1) the identification of minimal sets of model elements that are sufficient to prove a

---

given set of safety properties, as well as the subset of design elements that are necessary to prove the given properties; (2) the computation of minimal sets of design constraints whose violation leads the system to falsify one of more of the given properties.

Although these pieces of information are closely related, as we explain later, each of them can be naturally mapped to a typical use case in model-based software development: respectively, *merit assignment* and *blame assignment*. With the former the focus is on assessing the quality of a system specification, tracking the safety impact of model changes, and assisting in the synthesis of optimal implementations. With the latter, the goal is to determine the tolerance and resilience of a system against faults or cyber-attacks.

In general, proof-based traceability information can be used to perform a variety of engineering analyses, including vacuity detection [14]; coverage analysis [8, 11]; impact analysis [16], design optimization; and robustness analysis [18, 20]. Identifying which model elements are required for a proof, and assessing the relative importance of different model elements is critical to determine the quality of the overall model (including its assume-guarantee specification), determining when and where to implement changes, identifying components that need to be reverified, and measure the tolerance and resilience of the system against faults and attacks.

2 Preliminaries

Lustre is a synchronous dataflow language that allows one to define system components as *nodes*, each of which maps a continuous stream of inputs (of various basic types, such as Booleans, integers, and reals) to continuous streams of outputs based on both current input values and previous input and output values. Bigger components can be built by parallel composition of smaller ones, achieved syntactically with *node applications*. Operationally, a node has a cyclic behavior: at each tick t of a global clock (or a local clock it is explicitly associated with) it reads the value of each input stream at position or *time t*, and instantaneously computes and returns the value of each output stream at time t.

The behavior of a Lustre node is specified declarative by a set of stream constraints of the form $x = s$, where x is a variable denoting an output or a locally defined stream and s is a stream algebra over input, output, and local variables. Most stream operators are point-wise liftings of the usual operators over stream values. For example, if x and y are two integer streams, the expression $x + y$ is the stream corresponding the function $\lambda t.x(t) + y(t)$ over time t; an integer constant c, denotes the constant function $\lambda t.c$. Two important additional operators are a unary right-shift operator `pre`, used to specify stateful computations, and a binary initialization operator `->`, used to specify initial state values. At time $t = 0$, the value $(\texttt{pre } x)(t)$ is undefined; for each time $t > 0$, it is $x(t-1)$. In contrast, the value $(x \texttt{ -> } y)(t)$ equals $x(t)$ for $t = 0$ and $y(t)$ for $t > 0$. Syntactic restrictions guarantee that all streams in a node are inductively well defined. In Kind 2's extension of Lustre, nodes can be given assume-guarantee contracts, enabling the compositional analysis of Lustre models. Contracts specify assumptions as Boolean terms over current values of input streams and previous values of input and output streams, and guarantees as Boolean terms over current and previous values of input and output streams.

After various transformations and slicing, KIND 2 encodes Lustre nodes internally as state transition systems $S = \langle \mathbf{s}, I[\mathbf{s}], T[\mathbf{s}, \mathbf{s}'] \rangle$ where $\mathbf{s}$ is a vector of typed state variables, I the initial state predicate, and T is a two-state transition predicate (with $\mathbf{s}'$ being a renamed version of $\mathbf{s}$). We will use $\langle I, T \rangle$ to refer to transition system S when the vector of state variables $\mathbf{s}$ is clear from

the context or not important. A *state property* P for a system $S = \langle \mathbf{s}, I[\mathbf{s}], T[\mathbf{s}, \mathbf{s}'] \rangle$, expressed as a predicate over the variables $\mathbf{s}$, is *invariant* if it holds in every reachable state of S. We will say that S *satisfies* P, written as $S \models_I P$, if P is invariant for S. A *contract* for S is a pair $C = \langle A[\mathbf{s}], G[\mathbf{s}] \rangle$ where, informally, the *assumption* predicate A describes properties that S expects its inputs to have at all times, while the *guarantee* predicate G expresses how the component behaves when A does hold at all times.

In KIND 2, verifying that S satisfies its contract reduces in essence to verifying that $G[\mathbf{s}]$ is invariant for the system $S_A = \langle \mathbf{s}, I[\mathbf{s}] \wedge A[\mathbf{s}], A[\mathbf{s}] \wedge T[\mathbf{s}, \mathbf{s}'] \wedge A[\mathbf{s}'] \rangle$. For hierarchical and multi-component systems, KIND 2 translates the system model into a hierarchy of transition systems, where a call to a node N_B from a node N_A is represented by the assertion of the initial (transition) predicate of N_B in the initial (transition) predicate of N_A using the correspoding instantiation of the formal parameters. Transition systems provide then a convenient framework not only to check invariant properties but also to map and refer to different high-level specification and design elements in a uniform way. Given a transition system $\langle I, T \rangle$, we will assume that T has the structure of a top-level conjunction, that is, $T[\mathbf{s}, \mathbf{s}'] = T_1[\mathbf{s}, \mathbf{s}'] \wedge \ldots \wedge T_n[\mathbf{s}, \mathbf{s}']$ for some $n \geq 1$. Notice that this is the norm in Lustre models where the modeled system is expressed as the synchronous product of several subcomponents, each of which is in turn formalized as the conjunction of one or more equational constraints. Also, Kind 2's assume-guarantee contracts follow naturally this kind of conjunctive structure since they are specified as conjunction of assumptions and a conjunction of guarantees. By abuse of notation, we will identify T with the set $\{T_1, \ldots, T_n\}$ of its top-level conjuncts.

3 Running Example

We will use a simple model to illustrate the concepts and the functionality of KIND 2 introduced in this report. Suppose we want to design a component for an aircraft that controls the pitch motion of the vehicle, and suppose one of the system requirements is that the aircraft should not ascend beyond a certain altitude. The controller must read the current altitude of the aircraft from a sensor, and modify the next position of the aircraft's nose accordingly. For the sake of simplicity, we will ignore other relevant signals that should be considered in a real setting to control the elevation of the aircraft. Following a model-based design, we model an abstraction of the system's environment to which the aircraft's controller will react. We also model the fact that the system relies on a possibly imperfect reading of the current altitude by an altimeter sensor to decide the next pitch value. Finally, we provide a specification for the controller's behavior so that it satisfies the system requirement of interest.

Our model is described in Figure 1 in Kind 2's input language. The main node, `SystemModel`, is an *observer* node that represents the full system consisting in this case of just two components: one modeling the controller and a node modeling the environment. The observer has an input `sensor_alt` representing the altitude value from the altimeter and an output `actual_alt` representing the current altitude of the aircraft, which we are modeling as a product of the environment in response to the pitch value generated by the controller. Of course, in an actual aircraft that value would be communicated to an actuator, such as an elevator. We are exposing it directly in the observer's interface to simplify the specification since we are only interested in the relationship between the actual altitude and the pitch value.

KIND 2 allows the user to specify contracts for individual nodes, either as special Lustre comments added directly inside the node declaration, or as the instantiation of an external stand-alone

```
 1 node SystemModel (const THRESH, DELTA, S_ERROR: real;
 2                    sensor_alt: real) returns (actual_alt: real);
 3 (*@contract
 4   assume "C1: THRESH is positive" THRESH > 0.0;
 5   assume "C2: DELTA is positive" DELTA > 0.0;
 6   assume "C3: S_ERROR is non-negative" S_ERROR >= 0.0;
 7   assume "S: The error in the measured altitude is bounded"
 8     abs(0.0 -> pre actual_alt - sensor_alt) <= S_ERROR;
 9   guarantee "R1: Altitude is never above THRESH" actual_alt <= THRESH;
10 *)
11   var pitch: real;
12 let
13   pitch = Controller(THRESH, DELTA, S_ERROR, sensor_alt);
14   actual_alt = Environment(DELTA, pitch);
15 tel
16
17 node imported Controller (const THRESH, DELTA, S_ERROR: real;
18                           alt: real) returns (pitch: real);
19 (*@contract
20   const LIMIT: real = THRESH - (DELTA + S_ERROR);
21   guarantee "L1: Pitch is negative whenever altitude value is over LIMIT"
22     alt > LIMIT => pitch < 0.0;
23 *)
```

Fig. 1: System model and subcomponents. Operators `abs` and `=>` are respectively the absolute value function and Boolean implication.

contract that can be imported in the body of other contracts. The contract of `SystemModel`, included directly in the node, specifies assumptions on the altitude value provided by the sensor and on a number of symbolic constants (`THRESH`, `DELTA` and `S_ERROR`) which act in effect as model parameters. The contract assumes on lines 4–6 of Figure 1 that those constants are positive—or non-negative for `S_ERROR`. The assumption on line 7 accounts for fact that, while the altitude value produced by the altimeter (`sensor_alt`) is not 100% accurate in actual settings, its error is bounded by a constant (`S_ERROR`). The contract includes a guarantee (on line 9) that formalizes the requirement that aircraft maintain its altitude below a certain threshold `TRESH` at all times. The body of `SystemModel` is simply the parallel composition of the controller component with the environment node.

We do not specify the body of the `Controller` and the `Environment` nodes in our model because their details are not important for our purposes. Instead, we abstract their dynamics with an assume-guarantee contract that captures the relevant behavior. In the `Controller`'s case, we model the guarantee that the controller will produce a negative pitch value whenever the sensor altitude indicates that the aircraft is getting too close to the threshold value `THRESH` — with "too close" meaning that the difference between the current altitude and the threshold is smaller than `DELTA` + `S_ERROR` where `DELTA` represents an upper bound on the change in altitude from one execution step to the next (see below).

The declaration of the `Environment` component and its contract are shown separately in Figure 2. With `alt` representing the actual altitude of the aircraft, the contract's guarantees capture salient constraints on the physics of our model by specifying that a positive pitch value (which has the effect of raising the nose of the aircraft and lowering its tail) makes the aircraft ascend, a negative

```
1  node imported Environment (const DELTA: real;
2                             pitch: real) returns (alt: real);
3  (*@contract
4    guarantee "E1: Altitude is zero initially"
5      (alt = 0.0) -> true;
6    guarantee "E2: Altitude is always non-negative"
7      alt >= 0.0;
8    guarantee "E3: Altitude does not increase whenever the controller outputs a negative pitch
         value"
9      true -> (pitch < 0.0 => alt <= pre alt);
10   guarantee "E4: Altitude does not decrease more than DELTA units per sampling frame"
11     true -> (pitch < 0.0 => alt >= pre alt - DELTA);
12   guarantee "E5: Altitude does not decrease whenever the controller outputs a positive pitch
         value"
13     true -> (pitch > 0.0 => alt >= pre alt);
14   guarantee "E6: Altitude does not increase more than DELTA units per sampling frame"
15     true -> (pitch > 0.0 => alt <= pre alt + DELTA);
16   guarantee "E7: Altitude remains the same whenever the controller outputs a zero pitch value"
17     true -> (pitch = 0.0 => alt = pre alt);
18 *)
```

Fig. 2: Contract specification for the `Environment` component of `SystemModel`.

value makes it descend, and a zero value keeps it at the same altitude.[3] The contract also states that the actual altitude starts at zero, is alway non-negative, and does not change by more than a constant value (`DELTA`) in one sampling frame, where a sampling frame is identified with one execution step of the synchronous model for simplicity.[4]

KIND 2 can easily prove that property (guarantee) `R1` of `SystemModel` is invariant. However, a few interesting questions arise:

1. Is property `R1` satisfied because of the conditions we imposed on the behavior of `Controller`, or does the property trivially hold due to the stated assumptions over the environment and the sensor?
2. Are all the assumptions over the environment and the sensor in fact necessary to prove the satisfaction of property `R1`?
3. How resilient is the system against the failure of one or more assumptions?

We present the new features of KIND 2 that help us answer these questions next.

4 The New Features

The first of the two new features offered by KIND 2 consists in identifying which parts of the input model were used to construct an inductive proof of invariance for `R1`. The new functionality relies on the concept of inductive validity core introduced by Ghassabani et al. [10]. For the rest of the section, let us fix for convenience a transition system $S = \langle I, T \rangle$ and an invariant property P for S.

Definition 1 *A subset $C \subseteq T$ is an* inductive validity core *(IVC) for P if $\langle I, C \rangle \models_\mathrm{I} P$.*

---

[3] We are ignoring here that, in reality, altitude also depends on aircraft speed.
[4] The latter constraint captures physical limitations on the speed of the aircraft.

Note that, as P is invariant for S (i.e., $\langle I, T \rangle \models_I P$), T is already an IVC, although not a very interesting one. In practice, it is often possible to compute efficiently a smaller IVC that contains fewer or no irrelevant elements (see Section 5 for more details). We can ensure that the elements of an IVC for a property P are necessary by requiring it to be *minimal*, that is, to have no proper subset that is also IVC for P.

One can see every IVC as an *approximate* minimal IVC (MIVC). Computing approximate MIVCs is the recommended option in KIND 2 for quickly detecting modeling issues such as various forms of vacuity due to inconsistent assumptions or transition relation. KIND 2 can compute efficiently a single MIVC for a property P (see Section 5 for a brief description of the method, based on one by Ghassabani et al. [10]) and offers the option to compute all MIVCs for P.

4.1 IVCs for coverage and change impact analysis

If a property P of a system S has multiple MIVCs, inspecting all of them provides insights on the different ways S satisfies P. Moreover, given all the MIVCs for P, it is possible to partition all the model elements into three sets: a *MUST* set of elements which are required for proving P in every case, a *MAY* set of elements which are optional, and a set of elements that are irrelevant. More formally, let $MIVCs(S, P)$ denote the set of all MIVCs for S and P. Then, we have the following categorization [16]:

- $MUST(S, P) = \bigcap MIVCs(S, P)$
- $MAY(S, P) = (\bigcup MIVCs(S, P)) \setminus MUST(S, P)$
- $IRR(S, P) = T \setminus (\bigcup MIVCs(S, P))$

This categorization provides complete traceability between specification and design elements, and can be used for coverage analysis [11] and tracking the safety impact of model changes. For instance, a change to one of the elements in $MAY(S, P)$ will not affect the satisfaction of P but will definitely impact some other property Q if it occurs in $MUST(S, Q)$. Furthermore, we can use the set $IRR(S, P_1 \wedge \ldots \wedge P_m)$ of irrelevant elements for the conjunction of all the invariant properties $P1, \ldots, P_m$ of S to determine their completeness: if the set is non-empty that indicates that there may be missing requirements.

Example 1. If we ask KIND 2 to generate an approximate MIVC for the invariant R1 of the system presented in Section 3, KIND 2 generates a IVC with 7 elements: assumptions S and C1 from SystemModel's contract, the (only) guarantee L1 in Controller's contract, and all guarantees in Environment's contract except for E2, E4, and E5.

This tells us already that E2, E4, and E5 are not necessary to satisfy property R1 and is enough to answer the second of the questions listed at the end of Section 3. Moreover, since the guarantee L1 of Controller is part of the IVC, it is likely that the controller's behavior is relevant for the satisfaction of R1. However, we can not be sure because the generated IVC is an *approximate* MIVC and so is not necessarily minimal.

To confirm that L1 is indeed necessary we can ask KIND 2 to identify a true MIVC, a more expensive task computationally. When we do that, KIND 2 returns the same set. This confirms the necessity of the guarantee L1 but only for the specific proof of R1's invariance found by Kind 2. It might still be the case that the guarantee is not required in general, that is, there may be *other* proofs that do not use L1, which would be confirmed by the discovery of a different MIVC that does not contain it. In other words, at this point we do not know whether L1 is a *must* element for

```
1  node SystemModel (const THRESH, DELTA, S_ERROR: real;
2                    alt1, alt2, alt3: real) returns (actual_alt: real);
3  (*@contract
4    assume "C1: THRESH is positive" THRESH > 0.0;
5    assume "C2: DELTA is positive" DELTA > 0.0;
6    assume "C3: S_ERROR is non-negative" S_ERROR >= 0.0;
7    assume "S1: Error in altitude from sensor 1 is bounded by S_ERROR"
8      abs(0.0 -> pre actual_alt - alt1) <= S_ERROR;
9    assume "S2: Error in altitude from sensor 2 is bounded by S_ERROR"
10     abs(0.0 -> pre actual_alt - alt2) <= S_ERROR;
11   assume "S3: Error in altitude from sensor 3 is bounded by S_ERROR"
12     abs(0.0 -> pre actual_alt - alt3) <= S_ERROR;
13   guarantee "R1: Altitude never above THRESH" actual_alt <= THRESH;
14 *)
15   var pitch, alt: real;
16 let
17   alt = TriplexVoter(alt1, alt2, alt3);
18   pitch = Controller(THRESH, DELTA, S_ERROR, alt);
19   actual_alt = Environment(DELTA, pitch);
20 tel
```

Fig. 3: Enhanced system model.

R1. To determine that, and also to know if the assumptions not included in the computed MIVC are always irrelevant for the satisfaction of R1, we can ask KIND 2 to compute *all* MIVCs. In that case, KIND 2 will return only one MIVC, the set found in the first analysis, which confirms that all the included elements are required and the excluded ones are irrelevant. In alternative, we could have asked KIND 2 to compute a single MIVC *and* the MUST set for property R1. As we show later in this report, KIND 2 uses a method for generating the MUST set that does not require the computation of all MIVCs.   □

4.2 IVCs for fault-tolerance or cyber-resiliency analysis

Another use of IVCs, and MUST sets in particular, is in the analysis of a system's tolerance to faults [20] or resiliency to cyberattacks [18]. For instance, an empty MUST set for a system S and its invariant P indicates that the property is satisfied by S in various ways, making the system fault tolerant or resilient against cyberattacks as far as property P is concerned. In contrast, a large MUST set, as in our running example, suggest a more brittle system, with multiple points of failure or a big attack surface.

Example 2. Our previous analysis on the system model of our running example confirms that the correctness of the altitude sensor is crucial for the system not to exceed the prescribed altitude limit. One way to improve the system fault-tolerance then is to introduce some redundancy. In particular, we could equip the system with three different altimeters and so send to the controller three independent altitude values. The controller, or a dedicated new subcomponents could then take the average of the two altitude values that are closest to each other (as they more likely to be close to the actual altitude). This way, if one of the altimeter fails, in the sense that it produces an altitude reading with an error greater than the maximum expected error, the other two values allows the system to compensate for that error. We can easily change the model to implement

```
1  node TriplexVoter (alt1,alt2,alt3: real) returns (r: real);
2    var ad12,ad13,ad23,m: real;
3  let
4    ad12 = abs(alt1 - alt2);  ad13 = abs(alt1 - alt3);
5    ad23 = abs(alt2 - alt3);
6
7    m = min(ad12, min(ad13, ad23));
8
9    r = if m = ad12 then (alt1 + alt2) / 2.0
10       else if m = ad13 then (alt1 + alt3) / 2.0
11       else (alt2 + alt3) / 2.0;
12 tel
```

Fig. 4: Low-level specification of the Triplex voter. Operator `min` computes the minimum of its two inputs.

this redundancy mechanism. First, we extend the interface of `SystemModel` to take three altitude values (`alt1`, `alt2`, and `alt3`) instead of one, and then we introduce a new component, a triplex voter that takes those sensor values and computes an estimated altitude for the controller as explained above. The new specification for `SystemModel` is provided in Figure 3 with an updated contract for `SystemModel` that independently assumes the same error bounds on each individual altitude value. A full specification for the triplex voter is given in Figure 4.

We can use KIND 2 to confirm that property `R1` still holds after the introduction of the two new sensors and the triplex voter. However, this result is not enough to determine whether the introduced redundancy mechanism makes the system more fault tolerant. To confirms that we must verify the existence of multiple MIVCs. Perhaps surprisingly though, when we ask KIND 2 to compute all the MIVCs, it still reports a single solution—which includes all the sensor assumptions. Put differently, the MIVC analysis, shows that the satisfaction of property `R1` requires *all three* sensors to behave accordingly to their specification. After reviewing the model, however, one can conclude that to benefit from the triplex voter it is necessary to decrease the safety limit value `LIMIT` in the controller's contract. In particular, it is enough to decrease it as follows, doubling the error bound value:

```
1    const LIMIT: real = THRESH - (DELTA + 2.0*S_ERROR);
```

After this change, the number of MIVCs reported by KIND 2 increases from one to three. Each MIVC contains two of the assumptions `S1`, `S2`, `S3` on the three sensors, and the rest of the assumptions and guarantees included in the MIVC computed for the previous version of the model. This illustrates how the new traceability feature in KIND 2 could be used to detect a subtle flaw in the enhanced model that prevents it from making the system fault-tolerant despite the triplication of the altitude sensors. We stress how a simple safety analysis, verifying the invariance of `R1` would not help detect such flaw. □

4.3   Quantifying a system's resilience

To help quantify the resilience of a system, KIND 2 also supports the computation of minimal cut sets (aka, *minimal correction sets*) for an invariance property.

Definition 2 *A subset* $C \subseteq T$ *is a* cut set *for* P *if* $\langle I, T \setminus C \rangle \not\models_I P$. *A* minimal cut set *(MCS) for* P *is a cut set none of whose proper subsets is a cut set for* P. *A* smallest cut set *is an MCS of minimum cardinality. We will use* $MCSs(S, P)$ *to denote the set of all MCS for* S *and* P.

KIND 2 provides options to compute a (single) smallest cut set, all the MCSs, and all the MCSs up to a given cardinality bound. In the context of fault or security analyses, the cardinality of an MCS for P represents the number of design elements that must fail or be compromised for the property to be violated. The smaller the MCS, or the higher the number of MCSs of small cardinality, the greater the probability that the property can be violated.

It is worth mentioning that there exists a hitting set duality between MCSs and MIVCs, reflected by the following proposition, which provides a method to compute all MCSs from the set of all MIVCs:

Proposition 1 (Theorem 1 in [3]) *A subset* $C \subseteq T$ *is an MIVC for* P *iff* C *is a minimal hitting set of* $MCSs(S, P)$.

Notwithstanding the result above, there is a more direct approach, implemented in KIND 2 and described in Section 5, for computing all the MCSs which has the benefit of generating MCSs incrementally, without having to wait for the computation of all MIVCs. Similarly, there is a method for generating $MUST(S, P)$ that does not require the computation of all MIVCs. The method is based on the observation that $MUST(S, P)$ consists of the union of all MCSs of cardinality 1 for S and P.

Proposition 2 $MUST(S, P) = \{e \mid \{e\} \in MCSs(S, P)\}$.

KIND 2 offers the option to compute the MUST set together with a single MIVC or all of them. As the experiments of Section 6 show, the overhead of computing the MUST set in addition to an MIVC is negligible. Furthermore, KIND 2 uses the computation of the MUST set to check whether there exists only one MIVC, and terminate the search for more MIVCs early. The check is based on the result.

Proposition 3 *If* $MUST(S, P)$ *is an IVC then it is minimal and unique, i.e.,* $MIVCs(S, P) = \{MUST(S, P)\}$.

When looking for all MIVCs, KIND 2 always computes the set of MUST elements first and then checks whether $MUST(S, P)$ is enough to prove P. If that is the case, it stops looking for more MIVCs. The experiments in Section 6 indicate that this is an effective strategy.

Example 3. If we ask KIND 2 to compute all the MCSs for the version of the running example that includes the redundancy mechanism, KIND 2 finds the following MCSs: {E1}, {E3}, {E6}, {E7}, {C1}, {L1}, {S1, S2}, {S1, S3}, {S2, S3}.

5   Implementation details

We now give a high-level description of the algorithms implemented in KIND 2 to provide the functionality described in Section 4. Again, we fix a transition system $S = \langle I, T \rangle$ with $T = \{T_1, \ldots, T_n\}$ and an invariant property P for S.

The main algorithms use a number of auxiliary procedures, namely, Verify, MinimizeIVC, GetMCS, and GetApproximateMIVC, which are non-terminating in general for being based on checking the invariance of state properties of transition systems, an undecidable problem in the infinite-state case. In our concrete implementation, we make the main algorithms terminating by imposing a time limit on these procedures and handling timeout exceptions as follows. For Verify, we extend the type of the returned result with an additional value, `unknown`, to account for the exception. For MinimizeIVC, GetMCS, and the main algorithms, we have them return also a Boolean value indicating whether the result is precise or approximate due to a timeout. For GetApproximateMIVC, which calls Verify and then uses algorithm IVC_UC from [10], we have the procedure simply return T if Verify returns `unknown`, or the result of IVC_UC otherwise.

5.1 Computing approximate MIVC and single MIVC

The computation of an approximate MIVC is based on algorithm IVC_UC by Ghassabani et al. [10]. It consists of three main steps: (i) reducing the value of k for the k-inductive proof of property P (obtained by finding a k-inductive strengthening $Q = Q_1 \wedge \ldots \wedge Q_n$ of P); (ii) reducing the number of conjuncts in invariant Q by removing those not needed in the proof; (iii) computing an UNSAT core over the model constraints in the same query to the backend SMT solver that checks that Q is a k-inductive strengthening of P.

The computation of a single MIVC is based on algorithm IVC_UCBF, also by Ghassabani et al. [10]. The main idea is to generate an *approximate* MIVC first, and then minimize it using a brute-force approach that removes one model element at a time and (model) checks that the property P still holds.

5.2 Computing all MIVCs

To compute all MIVCs for S and P we adapted algorithm UMIVC by Berryhill and Veneris [3] which in turn is a generalization of previous work [1, 12]. Our version, described in Algorithm 1, starts by generating $MUST(S, P)$ (line 2) which is then used (line 4) to provide an initial value for variable *map*. That variable stores a CNF formula that tracks the portions of the power set of T explored so far. Each satisfying assignment for *map* corresponds to an abstraction of T that includes an element T_i exactly if variable a_i is true. The value of *ap* in line 2, which indicates $MUST$ is an underapproximation due to a timeout, is ignored because it does not affect soundness or completeness. Line 6 checks whether $MUST$ is an IVC. If it is, there exists only one MIVC and it corresponds to $MUST(S, P)$ (see Proposition 3). Otherwise, lines 9-20 compute all MIVCs. Line 10 extracts an abstraction of T from the map called *seed* of maximum cardinality using a MaxSAT solver. If the *seed* is not an IVC or the result is unknown, line 17 computes an MCS over $T \setminus seed$ that is used to prune the search space of candidates in line 18. The new clause forces any candidate solution to include at least one element of the MCS. Treating an unknown result as unsafe is sound, but it may affect completeness. In that case, *approx* is set to true in line 19. If the *seed* is an IVC, line 13 reduces the *seed* to an MIVC using an optimized version of algorithm IVC_UCBF that takes into account the fact that none of the elements in $MUST$ can be removed. If the minimization fails, the solution is tagged as approximate in line 14. Then, line 15 adds a clause to *map* that blocks the MIVC and all its supersets. The algorithm returns the computed set and a boolean value that is true if the solution is approximate, that is, there is no guarantee that every MIVC is contained in some element of the set.

Algorithm 1 AllMIVCs($S = \langle \mathbf{s}, I, T \rangle$, P)

1: $MIVCs := \emptyset$; $approx := \texttt{false}$
2: $MUST, ap := \text{MUST-SET}(\mathbf{s}, S, P)$
3: $A := \{a_i \mid 1 \leq i \leq |T|\}$ ▷ Fresh bool variables
4: $map := \bigwedge_{T_i \in MUST} a_i$
5: $res, \theta := \mathsf{Verify}(I, MUST, P)$
6: **if** $res = \texttt{safe}$ **then**
7:     $MIVCs := \{MUST\}$
8: **else**
9:     **while** $\mathsf{IsSAT}(map)$ **do**
10:         $seed := \mathsf{GetUnexploredMax}(map, T)$
11:         $res, \theta := \mathsf{Verify}(I, seed, P)$
12:         **if** $res = \texttt{safe}$ **then**   ▷ $seed$ is an IVC
13:             $mivc, ap := \mathsf{MinimizeIVC}(seed, MUST, S, P)$
14:             $MIVCs := MIVCs \cup \{(mivc, ap)\}$
15:             $map := map \wedge \bigvee_{T_i \in mivc} \neg a_i$
16:         **else**
17:             $mcs, ap := \mathsf{GetMCS}(S, T \setminus seed, P)$
18:             $map := map \wedge \bigvee_{T_i \in mcs} a_i$
19:             $approx := approx \vee res = \texttt{unknown}$
20:     **end while**
21: **return** $(MIVCs, approx)$

Algorithm 1 can be seen as an instantiation of UMIVC where all MCSs of cardinality 1 are precomputed. The major difference with UMIVC is that our algorithm is able to identify the MUST set from the generated set of MCSs (c.f. Proposition 2), and can use it to check for early termination and to boost the minimization of the IVC in line 13. Our current implementation in KIND 2 is in fact a generalization of Algorithm 1 that allows the user to choose the set of model elements (assumptions and guarantees, node calls, equations in node bodies, ...) over which the MIVCs must be computed. It assumes that the rest of model elements are always present in the system.

5.3 Computing all MCSs

Given a subset E of $T = \{T_1, \ldots, T_n\}$, KIND 2 uses an implementation of Algorithm 2 to find all the MCSs over E for S and P with cardinality not greater than a given upper bound ub. The algorithm can be used to compute all MCSs for S and P by choosing E to be T and ub to be $|T|$.

The algorithm is based on techniques previously applied to automated debugging [2, 19]. To understand how it works it is helpful to observe that we can reduce the problem of finding one cut set for S and P to a model checking problem. Specifically, we build a modified version of S, $S^\star = \langle \mathbf{z}, I[\mathbf{z}], T^\star[\mathbf{z}, \mathbf{z}'] \rangle$, where the vector of typed variables $\mathbf{z}$ includes $\mathbf{s}$ and fresh Boolean variables $\mathbf{y} = \langle y_1, \ldots, y_{|T|} \rangle$, and the transition predicate is defined by $T^\star[\mathbf{z}, \mathbf{z}'] = \bigwedge_{T_i \in T}(\neg y_i \Rightarrow T_i \wedge y_i' = y_i)$. Then, we check whether $S^\star$ satisfies P or not.

If we can disprove P, the initial values assigned to $\mathbf{y}$ in any trace that leads $S^\star$ to the violation of P determines a cut set: the set of all T_i's such as the initial value of y_i is true. Moreover, if we add a cardinality constraint over $\mathbf{y}$ in I specifying that at most k of the variables are true, any solution C will be such that $|C| \leq k$. Lines 6-11 use this reduction to find an MCS of minimal cardinality. If none exists, or the first call to Verify returned unknown, the condition in line 13 is false, and

Algorithm 2 AllMCSs_UpToUB($S = \langle \mathbf{s}, I, T \rangle$, E, P, ub)

1: $MCSs := \emptyset$; $approx := \mathtt{false}$
2: $m := min(ub, |E|)$
3: $\mathbf{y} = \langle y_1, \ldots, y_{|E|} \rangle$ ▷ Fresh bool variables
4: $T^\star := \bigwedge_{T_i \in E}(\neg y_i \Rightarrow T_i \wedge y_i' = y_i) \wedge \bigwedge_{T_j \in (T \setminus E)} T_j$
5: $k := m$; $res := \mathtt{unknown}$; $\theta := \emptyset$
6: **do**
7:      $I^\star := I \wedge \mathsf{AtMostK}(\mathbf{y}, k)$
8:      $res, \theta' := \mathsf{Verify}(I^\star, T^\star, P)$
9:      **if** $res = \mathtt{unsafe}$ **then**
10:          $\theta := \theta'$; $k := k - 1$ ▷ Store last counterexample, and decrement k
11: **while** $k \geq 0 \wedge res = \mathtt{unsafe}$
12: $approx := approx \vee res = \mathtt{unknown}$
13: **if** $k < m$ **then**   ▷ Unsafe for $k + 1$
14:      **if** $k < 0$ **then** $MCSs := \{\emptyset\}$ ▷ No Safe for any k
15:      **else**
16:          $k := k + 1$ ▷ Cardinality of a smallest MCS
17:          $C := \mathsf{ExtractCutSet}(\theta, \mathbf{y}, E)$
18:          $MCSs := MCSs \cup \{(C, approx)\}$
19:          $\phi := \bigvee_{T_i \in C} \neg y_i$ ▷ Block supersets of C
20:          **while** $k \leq m$ **do**
21:              $I^\star := I \wedge \mathsf{AtMostK}(\mathbf{y}, k)$
22:              $res, \theta' := \mathsf{Verify}(I^\star \wedge \phi, T^\star, P)$
23:              **while** $res = \mathtt{unsafe}$ **do**
24:                  $C := \mathsf{ExtractCutSet}(\theta', \mathbf{y}, E)$
25:                  $MCSs := MCSs \cup \{(C, approx)\}$
26:                  $\phi := \phi \wedge \bigvee_{T_i \in C} \neg y_i$
27:                  $res, \theta' := \mathsf{Verify}(I^\star \wedge \phi, T^\star, P)$
28:              **end while**
29:              $approx := approx \vee res = \mathtt{unknown}$
30:              $k := k + 1$
31:          **end while**
32: **return** $(MCSs, approx)$

---

the empty set is returned. If the original system S does not satisfy P, condition in line 14 is true, and the only MCS is the empty set. Otherwise, lines 17-31 compute all (non-empty) MCSs. The first MCS is extracted in line 17 from θ, which is the last error trace found in the do-while loop. To block the found MCS and all its supersets, line 19 initializes ϕ, a CNF formula which tracks which portion of the power set of E has been explored. After that, the rest of MCSs of cardinality k [5] are extracted in the internal loop (lines 23-28). When no more MCSs of cardinality k exists, k is incremented by one in line 30. If the maximum cardinality has not been reached yet, checked in line 20, the internal loop continues to extract all the MCSs of cardinality k. Since ϕ contains clauses that block any superset of an MCS of cardinality less than k, the new MCSs generated are guaranteed to be minimal if no call to Verify has returned unknown. Otherwise, minimality is not guaranteed, and the solution is tagged as approximate. Moreover, the algorithm returns true together the generated set to indicate that some MCS may not be contained in any element of the set.

---

[5] The minimal cardinality if *approx* has not been set to true.

Algorithm 3 MUST-Set($S = \langle \mathbf{s}, I, T \rangle$, P)

1: ivc := GetApproximateMIVC(S, P)
2: $MCSs$, $approx$:= ALLMCSs_UPTOUB(S, ivc, P, 1)
3: **return** ($\{e \mid (\{e\}, \texttt{false}) \in MCSs\}$, $approx$)

5.4 Computing the MUST set

We use Algorithm 3 to compute the MUST set for a transition system S and an invariant property P. It first finds an *approximate* MIVC, *ivc*, using algorithm IVC_UC from [10]. Then, it uses the *ivc* to narrow the search of MCSs of cardinality 1. When the result returned in line 2 is precise, it builds the MUST set from the generated MCSs relying on Proposition 2, and returns the set together the boolean value false. Otherwise, the algorithm returns an underapproximation [6] of the MUST set together the boolean value true. This algorithm shares some similarities with one by Chockler et al. [8] for determining non-deterministic mutation coverage (Nondet-Cov). This is expected since Nondet-Cov is equivalent to coverage based on MUST elements, as proved by Ghassabani et al. [11].

6 Experimental Evaluation

Using the set of benchmarks from [12], we ran two experiments[7] to address the following questions: (*i*) how the computation of both a single MIVC (using IVC_UCBF) and the MUST set (using Algorithm 3) compares performance-wise with the computation of just an MIVC, and (*ii*) what is the actual advantage of computing the MUST set to prune the search space and checking the existence of a single MIVC in Algorithm 1.

For question (*i*), our results, illustrated in Figure 5a, reveal that the overhead of generating the MUST set in addition to a single MIVC is negligible. However, the feedback provided to the users is highly informative as it allows them to quickly check if there is only one MIVC and, if not, determine how many elements of the MIVC are required. For question (*ii*), our results confirm that the optimization is beneficial as Algorithm 1 exhibits on average a 1.5x speedup over the version without the computation, as illustrated by the cactus plot in Figure 5b.

7 Related Work

The computation of an approximate MIVC was first available in the open-source model checker JKIND [9] around the same time the technique was introduced in [10]. More recently, JKIND started to offer support for the computation of all MIVCs based on the *offline* algorithm decribed in [12]. The algorithm is considered *offline* because it is not until all IVCs have been computed that one knows whether the solutions computed are, in fact, minimal. For models contain many IVCs, this approach can be impractically expensive or simply not terminate. However, for applications where only a full enumeration of the MIVCs is this technique may offer better overall performance. The main idea is to use algorithm IVC_UC for the minimization of the IVC in line 13 of Algorithm 1, as opposed to the more expensive algorithm IVC_UCBF that ensures minimality, and to not minimize the cut set in line 17 before adding it to the map.

---

[6] Notice that every returned cut set (of cardinality 1) is minimal if P is invariant.
[7] All materials and pointers to the new version of KIND 2 are available at https://github.com/kind2-mc/mivc-must-experiments.

(a) Computation of a single MIVC   (b) Computation of all MIVCs
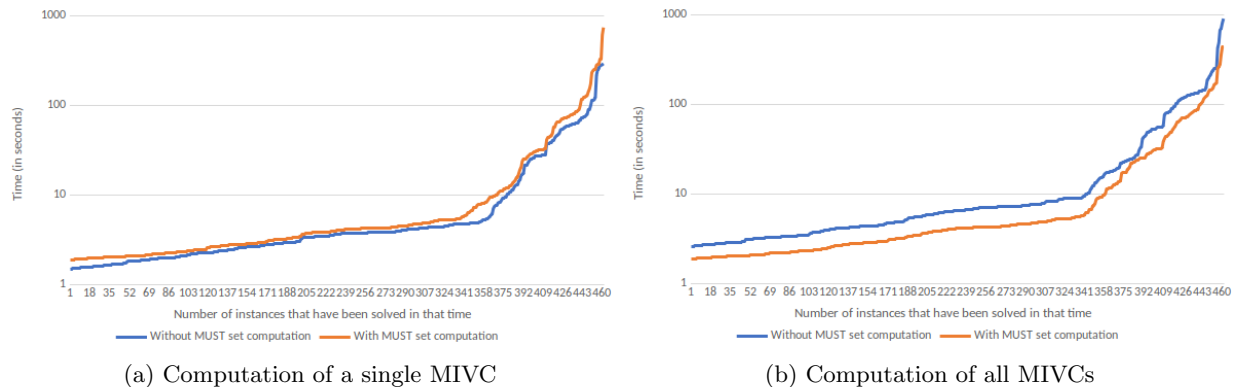
Fig. 5: Experimental Results

Although not part of the official distribution of JKIND, the *online* algorithm for computing all MIVCs presented in [1] has also been implemented in the tool. Similarly to Algorithm 1, it incorporates the idea of reducing the cardinality of the cut sets generated when calls to Verify returns unsafe. In contrast, the method only tries to reduce the cardinality when Verify returns unsafe within algorithm IVC_UCBF, not in the main loop of Algorithm 1. Moreover, the reduction is based on retrieving a maximal set of *map* that contains the seed, and checking whether the subset is a IVC or not. If it is not a IVC, the complement of the subset is an approximation of a MCS. Otherwise, approximate MIVCs are computed and used to reduce the elements of the seed until the subset is not a IVC anymore.

Unlike KIND 2, JKIND does not have native support for assume-guarantee contracts in its input language. Thus, JKIND only considers equations for the generation of IVCs. In contrast, KIND 2 allows the user to select not only design elements such as node calls and equations but also specification elements such as assumptions and guarantees.

An algorithm for computing all MCSs is described by Bozzano et al. [4]. Like our Algorithm 2, their technique computes the cuts sets of increasing cardinality to prevent the generation of non-miminal solutions. However, their method relies on a IC3-based routine for parameter synthesis to compute all the solutions in each layer. Therefore, instead of relying on a black-box Verify procedure to solve multiple ordinary model checking queries, they use a specialized algorithm. The main advantage in that case is that the information learnt to block a particular counterexample can be reused when considering new ones.

References

1. Bendík, J., Ghassabani, E., Whalen, M.W., Černá, I.: Online enumeration of all minimal inductive validity cores. In: Johnsen, E.B., Schaefer, I. (eds.) Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10886, pp. 189–204. Springer (2018). https://doi.org/10.1007/978-3-319-92970-5_12
2. Berryhill, R., Veneris, A.G.: Methodologies for diagnosis of unreachable states via property directed reachability. IEEE Trans. on CAD of Integrated Circuits and Systems **37**(6), 1298–1311 (2018). https://doi.org/10.1109/TCAD.2017.2747999
3. Berryhill, R., Veneris, A.G.: Chasing minimal inductive validity cores in hardware model checking. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019. pp. 19–27. IEEE (2019). https://doi.org/10.23919/FMCAD.2019.8894268

4. Bozzano, M., Cimatti, A., Griggio, A., Mattarei, C.: Efficient anytime techniques for model-based safety analysis. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 603–621. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_41

5. Bradley, A.R.: Sat-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7

6. Champion, A., Gurfinkel, A., Kahsai, T., Tinelli, C.: Cocospec: A mode-aware contract language for reactive systems. In: Nicola, R.D., eva Kühn (eds.) Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9763, pp. 347–366. Springer (2016). https://doi.org/10.1007/978-3-319-41591-8_24

7. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 510–517. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_29

8. Chockler, H., Kroening, D., Purandare, M.: Coverage in interpolation-based model checking. In: Sapatnekar, S.S. (ed.) Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010. pp. 182–187. ACM (2010). https://doi.org/10.1145/1837274.1837320

9. Gacek, A., Backes, J., Whalen, M., Wagner, L.G., Ghassabani, E.: The JKind model checker. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 20–27. Springer (2018). https://doi.org/10.1007/978-3-319-96142-2_3

10. Ghassabani, E., Gacek, A., Whalen, M.W.: Efficient generation of inductive validity cores for safety properties. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 314–325. ACM (2016). https://doi.org/10.1145/2950290.2950346

11. Ghassabani, E., Gacek, A., Whalen, M.W., Heimdahl, M.P.E., Wagner, L.G.: Proof-based coverage metrics for formal verification. In: Rosu, G., Penta, M.D., Nguyen, T.N. (eds.) Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017. pp. 194–199. IEEE Computer Society (2017). https://doi.org/10.1109/ASE.2017.8115632

12. Ghassabani, E., Whalen, M.W., Gacek, A.: Efficient generation of all minimal inductive validity cores. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 31–38. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102238

13. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. IEEE Trans. Software Eng. **18**(9), 785–793 (1992). https://doi.org/10.1109/32.159839

14. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. Int. J. Softw. Tools Technol. Transf. **4**(2), 224–233 (2003). https://doi.org/10.1007/s100090100062

15. Mebsout, A., Tinelli, C.: Proof certificates for smt-based model checkers for infinite-state systems. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016. pp. 117–124. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886669

16. Murugesan, A., Whalen, M.W., Ghassabani, E., Heimdahl, M.P.E.: Complete traceability for requirements in satisfaction arguments. In: 24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016. pp. 359–364. IEEE Computer Society (2016). https://doi.org/10.1109/RE.2016.35

17. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a sat-solver. In: Jr., W.A.H., Johnson, S.D. (eds.) Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1954, pp. 108–125. Springer (2000). https://doi.org/10.1007/3-540-40922-X_8

18. Siu, K., Moitra, A., Li, M., Durling, M., Herencia-Zapana, H., Interrante, J., Meng, B., Tinelli, C., Chowdhury, O., Larraz, D., et al.: Architectural and behavioral analysis for cyber security. In: 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC). pp. 1–10. IEEE (2019)

19. Smith, A., Veneris, A.G., Ali, M.F., Viglas, A.: Fault diagnosis and logic debugging using boolean satisfiability. IEEE Trans. on CAD of Integrated Circuits and Systems **24**(10), 1606–1621 (2005). https://doi.org/10.1109/TCAD.2005.852031

20. Stewart, D., Liu, J.J., Whalen, M.W., Cofer, D., Peterson, M.: Safety annex for the architecture analysis and design language (2020)