Merit and Blame Assignment with Kind 2

Daniel Larraz[1(✉)], Mickaël Laurent[1,2], and Cesare Tinelli[1]

[1] Department of Computer Science, The University of Iowa, Iowa City, USA
daniel-larraz@uiowa.edu

[2] IRIF, CNRS—Université de Paris, 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. We illustrate with an example how to use the computed information for tracking the safety impact of model changes, and for analyzing the tolerance and resilience of a system against faults.

Keywords: SMT-based model checking · Inductive validity cores · Traceability · MUST-set generation · Minimal Cut Sets

1 Introduction

KIND 2 [6] is an open-source[1] 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 [11]. The extension allows the specification of assume-guarantee-style contracts for the modeled system and its components which enables modular and compositional reasoning and considerably increases scalability. KIND 2's contract language [5] is expressive enough to allow one to represent any (LTL) regular safety property by recasting it in terms of invariant properties. 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 [16] and IC3 [4], 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, some model checkers are able to return some form of corroborating evidence when they declare a safety property to be satisfied by a system under analysis.

Work partially funded by DARPA grant #N66001-18-C-4006 and by GE Global Research.

[1] KIND 2 is distributed under the Apache 2.0 License at http://kind.cs.uiowa.edu.

For instance, KIND 2 can produce an independently checkable proof certificate for the properties that it claims to have proven [14]. 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 diagnostic 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 two pieces of information are closely related, 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 [12]; coverage analysis [7,9]; impact analysis [15], design optimization; and robustness analysis [17,18]. 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 Running Example

We will use a simple model to illustrate the concepts and the functionality of KIND 2 introduced in this paper. Suppose we want to design a component for an airplane that controls the pitch motion of the aircraft, 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. Moreover, we want the system to be fault-tolerant to sensor failures. One way to improve system fault-tolerance is to introduce some redundancy. In particular, we can equip the system with three different altimeters so the controller receives three independent altitude values. Then the controller, with the help of a dedicated component, a *triplex voter*, takes the average of the two altitude values that are closest to each other—as they are more likely to be close to the actual altitude. For 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 possibly imperfect readings of the current altitude by the sensors 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.

```
1  node SystemModel (const TH, UB, ERR: real; alt1, alt2, alt3: real)
2  returns (act_alt: real);
3  (*@contract
4    assume "C1" TH > 0.0; assume "C2" UB > 0.0; assume "C3" ERR >= 0.0;
5    assume "S1" abs(0.0 -> pre act_alt - alt1) <= ERR;
6    assume "S2" abs(0.0 -> pre act_alt - alt2) <= ERR;
7    assume "S3" abs(0.0 -> pre act_alt - alt3) <= ERR;
8    guarantee "R1" act_alt <= TH;
9  *)
10   var pitch, alt: real;
11 let
12   alt = TriplexVoter(alt1, alt2, alt3);
13   pitch = Controller(TH, UB, ERR, alt);
14   act_alt = Environment(UB, pitch);
15 tel
16
17 node imported Controller (const TH, UB, ERR: real; alt: real) returns (pitch: real);
18 (*@contract
19   const LIMIT: real = TH - (UB + ERR);
20   guarantee "L1" alt > LIMIT => pitch < 0.0;
21 *)
```

Fig. 1. System model and subcomponents. Operators −>, abs and => are respectively the initialization operator, the absolute value function, and Boolean implication.

Our model is described in Fig. 1 in KIND 2's input language where system components are called *nodes*. The main component, SystemModel, is an *observer* node that represents the full system consisting in this case of just three subcomponents: one node modeling the controller, one modeling a triplex voter, and another one modeling the environment. The observer has three inputs: alt1, alt2, and alt3, representing the altitude values from each altimeter, and an output act_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.

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 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 values provided by the sensors and on a number of symbolic constants (TH, UB and ERR) which act in effect as model parameters. The contract assumes at line 4 that those constants are positive, or non-negative for ERR. The assumptions at lines 5–7 account for fact that, while the altitude value produced by each altimeter is not 100% accurate in actual settings, its error is bounded by a constant (ERR)[2]. The contract

[2] The initialization operator −> is used to specify initial state values. Operationally, a node has a cyclic behavior: at each tick t of an abstract global clock it reads the value of each input stream at time t, and instantaneously computes the value of each output stream at time t. For streams x and y, the value $(x -> y)(t)$ for stream x -> y equals $x(t)$ for $t = 0$ and $y(t)$ for $t > 0$.

```
node TriplexVoter (alt1,alt2,alt3: real) returns (r: real);
  var ad12,ad13,ad23,m,avg1,avg2,avg3: real;
let
  (ad12, ad13, ad23) = (abs(alt1 - alt2), abs(alt1 - alt3), abs(alt2 - alt3));
  m = min(ad12, min(ad13, ad23));
  (avg1, avg2, avg3) = (alt1 + alt2) / 2.0, (alt1 + alt3) / 2.0, (alt2 + alt3) / 2.0));
  r = if m = ad12 then avg1 else if m = ad13 then avg2 else avg3;
tel
```

Fig. 2. Low-level specification of the Triplex voter.

```
node imported Environment (const UB: real; pitch: real) returns (alt: real);
(*@contract
  guarantee "E1" (alt = 0.0) -> true;
  guarantee "E2" alt >= 0.0;
  guarantee "E3" true -> (pitch < 0.0 => alt <= pre alt);
  guarantee "E4" true -> (pitch < 0.0 => alt >= pre alt - UB);
  guarantee "E5" true -> (pitch > 0.0 => alt >= pre alt);
  guarantee "E6" true -> (pitch > 0.0 => alt <= pre alt + UB);
  guarantee "E7" true -> (pitch = 0.0 => alt = pre alt);
*)
```

Fig. 3. Contract specification for the Environment component of SystemModel.

includes a guarantee (line 8) that formalizes the requirement that aircraft maintain its altitude below a certain threshold TH at all times. The body of SystemModel is simply the parallel composition of a triplex voter, that takes the sensor values and computes an estimated altitude for the controller as explained above, the controller component, and the environment node.

A full specification for the TriplexVoter is given in Fig. 2. 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 TH—with "too close" meaning that the difference between the current altitude and the threshold is smaller than UB + ERR where UB 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 Fig. 3. 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 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

[3] We are ignoring here that, in reality, the altitude also depends on aircraft speed.

zero, is alway non-negative, and does not change by more than a constant value (UB) in one sampling frame, where a sampling frame is identified with one execution step of the synchronous model (one global clock tick) for simplicity. The latter constraint on the altitude change rate captures physical limitations on the speed of the aircraft.

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 sensors? (2) Are all the assumptions over the environment and the sensors 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 in the following the new features of KIND 2 that help us answer these questions. A demo video associated to this paper can be found here [1].

3 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. [8]. Generally speaking, given a set of *model elements* M and an invariant property P, an *inductive validity core* (IVC) for P is a subset of M that is enough to prove P invariant. Kind 2 allows the user to choose among four sets of model elements: assumptions/guarantees, node calls, equations in node definitions[4], and assertions[5]. In our running example, we consider $P = \mathsf{R1}$ and $M = S_1 \cup C_2 \cup E_3 \cup \{\mathsf{L1}\}$ where $S_1 = \{\mathsf{S1}, \mathsf{S2}, \mathsf{S3}\}$, $C_2 = \{\mathsf{C1}, \mathsf{C2}, \mathsf{C3}\}$, and $E_3 = \{\mathsf{E1}, \mathsf{E2}, \mathsf{E3}, \mathsf{E4}, \mathsf{E5}, \mathsf{E6}, \mathsf{E7}\}$. In particular, note that M is an IVC, although not a very interesting one. In practice, for complex enough models, smaller IVCs exist. In particular is often possible to compute efficiently a smaller IVC that contains few or no irrelevant elements. 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. KIND 2 offers the option to compute a *small* but possibly non-minimal IVC, *a minimal* IVC (MIVC), or *all minimal* IVCs.

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 [15]: 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. This categorization provides complete traceability between specification and design elements, and can be used for coverage analysis [9] and tracking the safety impact of model changes. For instance, a change to one of the elements in the *MAY* set for P will not affect the satisfaction of P but will definitely impact some other property Q if it occurs in the *MUST* set for Q.

IVCs for Fault-Tolerance or Cyber-resiliency Analysis. Another use of IVCs, is in the analysis of a system's tolerance to faults [18] or resiliency to cyber-attacks [17].

[4] Note that a node is Lustre is defined declaratively by a set of equations.

[5] In Lustre, assertions are (unchecked) assumptions on a node's input.

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 cyber-attacks as far as property P is concerned. In contrast, a large MUST set suggest a more brittle system, with multiple points of failure or a big attack surface.

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. Given a set of model elements M and an invariant property P, a *cut set* C for P is a subset of M such that P is no longer invariant for $M \setminus C$. 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. 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 a property P represents the number of design elements that must fail or be compromised for P 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.

Running Example. If we ask KIND 2 to generate an IVC for the invariant R1 of the system presented in Sect. 2, KIND 2 generates a IVC with 9 elements: assumptions S1, S2, S3, and C1 from SystemModel's contract, the (only) guarantee L1 in Controller's contract, and all guarantees in the contract of Environment 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 Sect. 2. 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 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 R1. To determine that, we can ask KIND 2 to compute the MUST set for property R1 in addition to the MIVC. In that case, KIND 2 will return the same set as the MUST set, which confirms that all the included elements are required and the excluded ones are irrelevant.

Note that the last result also means that assumptions S1, S2, and S3 are always necessary, and thus, property R1 requires *all three* sensors to behave accordingly to their specification. Put differently, the analysis shows that the introduced redundancy mechanism *does not* actually make the system more fault tolerant. After reviewing the model, however, one can conclude that to benefit from the triplex voter we must decrease the safety limit value LIMIT in the controller's contract. Specifically, it is enough to decrease it as follows, doubling the error bound value:

```
const LIMIT: real = TH − (UB + 2.0 ∗ ERR);
```

After this change, KIND 2 stops classifying assumptions S1, S2, and S3 as MUST elements. It computes a new MIVC of 8 elements which differs from the one computed for the previous version of the model for the absence of S3. To confirm that this MIVC is not the only solution, we can ask KIND 2 to compute all the MIVCs instead of a single one. This makes KIND 2 show two additional MIVCs that are symmetric to the computed MIVC: one set that contains S1 and S3 rather than S1 and S2, and another one that contains S2 and S3 instead S1 and S2. In alternative, we could ask KIND 2 to compute all the MCSs for the revised model. In that case, KIND 2 will find the following MCSs: {E1}, {E3}, {E6}, {E7}, {C1}, {L1}, {S1,S2}, {S1,S3}, {S2,S3}. This confirms that the system can now tolerate the failure of one of its three altimeters.

The exercise above illustrates how the new traceability feature in KIND 2 could be used to detect a subtle flaw in our enhanced model that prevented 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 Implementation Details

KIND 2 is written in OCaml. All logical reasoning done by KIND 2 eventually reduces to queries to an external SMT solver. The implementation of the new features required around 2.8 KLOC. The computation of a small IVC for a property P is based on algorithm IVC_UC by Ghassabani et al. [8]. 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 $Q = Q_1 \wedge \cdots \wedge Q_n$ 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. [8]. The main idea is to generate a small IVC 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.

To compute all MIVCs we adapted algorithm UMIVC by Berryhill and Veneris [3] which in turn is a generalization of previous work [2,10]. It basically explores in an efficient way the power set of model elements. The algorithm implemented in KIND 2 can be seen as an instantiation of UMIVC where all MCSs of cardinality 1 are pre-computed. The major difference with UMIVC is that our algorithm is able to identify the MUST set from the generated set of MCSs, which can be use to check for early termination of the algorithm and to enhance the minimization of the intermediate IVCs generated during the process.

The problem of finding one cut set for a system S and a property P with at most k model elements is reduced to a model checking problem. Every violation of property P in this problem leads to a cut set. KIND 2 keeps solving this model checking problem using smaller and smaller bounds until there is no more violations. When that happens, it can extract a cut set of minimal cardinality. KIND 2 is also able to find all possible MCSs with cardinality smaller than a given bound by incrementally adding constraints that block the previous solutions. When there are no more minimal sets within the current cardinality bound, it increases that bound by one and repeats the process. It ends this process when the cardinality bound equals the number of model elements considered, having computed at that point all possible MCSs.

We refer the interested reader to a related technical report [13] for further implementation details and experimental results.

References

1. Demo video. https://doi.org/10.5281/zenodo.5070546. Accessed 5 July 2021
2. Bendík, J., Ghassabani, E., Whalen, M., Černá, I.: Online enumeration of all minimal inductive validity cores. In: Johnsen, E.B., Schaefer, I. (eds.) SEFM 2018. LNCS, vol. 10886, pp. 189–204. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92970-5_12
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, 22–25 October 2019. pp. 19–27. IEEE (2019). https://doi.org/10.23919/FMCAD.2019.8894268
4. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
5. Champion, A., Gurfinkel, A., Kahsai, T., Tinelli, C.: CoCoSpec: a mode-aware contract language for reactive systems. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 347–366. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_24
6. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The KIND 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 510–517. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_29
7. 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, 13–18 July 2010. pp. 182–187. ACM (2010). https://doi.org/10.1145/1837274.1837320
8. 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, 13–18 November 2016. ,p. 314–325. ACM (2016). https://doi.org/10.1145/2950290.2950346
9. 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, 30 October–03 November 2017, pp. 194–199. IEEE Computer Society (2017). https://doi.org/10.1109/ASE.2017.8115632
10. 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, 2–6 October 2017, pp. 31–38. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102238
11. 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
12. 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
13. Larraz, D., Laurent, M., Tinelli, C.: Merit and blame assignment with kind 2. CoRR abs/2105.06575 (2021). https://arxiv.org/abs/2105.06575

14. 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, 3–6 October 2016, pp. 117–124. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886669
15. 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, 12–16 September 2016, pp. 359–364. IEEE Computer Society (2016). https://doi.org/10.1109/RE.2016.35
16. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
17. Siu, K., et al.: Architectural and behavioral analysis for cyber security. In: 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), pp. 1–10. IEEE (2019)
18. Stewart, D., Liu, J.J., Whalen, M.W., Cofer, D., Peterson, M.: Safety annex for the architecture analysis and design language (2020)