Beyond model checking of idealized Lustre in Kind 2

Daniel Larraz, Arjun Viswanathan, Cesare Tinelli
The University of Iowa
USA

Mickaël Laurent
IRIF, CNRS — Université de Paris
France

Abstract

This paper describes several new features of the open-source model checker Kind 2. Its input language and model checking engines have been extended to allow users to model and reason about systems with machine integers. In addition, Kind 2 can now provide traceability information between specification and design elements, which 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. Finally, Kind 2 is also able to check whether a component contract is realizable or not, and provide a deadlocking computation and a set of conflicting guarantees when the contract is unrealizable.

Keywords: Machine-precise Model Checking, Safety Analysis, Realizability Checking

1 Introduction

Kind 2 [4] 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 dataflow 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 [3] 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 architectural 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 [20] and IC3 [2], with various auxiliary invariant generation methods. All the engines are fully automated and logic-based, relying on external SMT solvers for satisfiability/entailment checks and other relevant logical operations such as quantifier elimination.

Kind 2 is open-source and distributed in binary and source-code form under a liberal license.[1] This paper focuses on its most recent features, in particular, reasoning about models with machine integer values, providing traceability information between specification and design elements, and checking component contracts are realizable.

2 Machine-precise Verification

Lustre is a synchronous dataflow language that operates on infinite streams of values of three basic types: `bool`, `int` (finite precision integers), and `real` (floating point numbers). In contrast, Kind 2 considers an idealized version of Lustre, which treats `int` as the type of mathematical integers, and `real` as the type of real numbers. Idealized Lustre programs can be faithfully encoded 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 is the initial state predicate, and T is a two-state transition predicate (with $\mathbf{s}'$ being a renamed version of $\mathbf{s}$). Then, instances of I and T can be expressed as quantifier-free first-order formulas over the combined theory of equality with uninterpreted functions and integer/real arithmetic. SMT solvers implement efficient decision procedures for the fragment of this theory that limits arithmetic constraints to linear ones. Although using idealized Lustre is often adequate for proving and disproving a wide range of properties of programs of interest, sometimes it is important to reason with respect to the original semantics of the numeric types (for instance, to capture accurately the modulo n behavior of arithmetic operators over machine integers). For the latter cases, we have extended Kind 2 to support both signed and unsigned versions of C-style machine integers of size 8, 16, 32, and 64.

Semantics, declaration, and value construction. The standard semantics of machine integers of size w is binary numbers of width w, with signed machine integers represented using 2's complement. We effectively adopt the same semantics by representing machine integers internally as (signed or unsigned) bit vectors of width w. Kind 2 currently supports signed machine integers of width 8, 16, 32

[1] Kind 2 is available at http://kind.cs.uiowa.edu.

HILT'22, October 2022, Oakland, Michigan, USA

and 64, allowing expressions of types `int8`, `int16`, `int32`, and `int64`, respectively, and their unsigned versions, with types `uint8`, `uint16`, `uint32`, and `uint64`. Machine integers values can be constructed using explicit conversion functions applied to integer constants, with a conversion functions for each possible destination type. For example, `uint8` converts any numeral n to the unsigned 8-bit value corresponding to the integer value ($n \mod 8$). This means, for instance that `uint8 0` and `uint8 256` are both converted to the 8-bit zero value. Conversions in the opposite direction are also possible, with the expected inclusion semantics. Conversions between machine integers of different widths are also allowed as long as the types are both signed or both unsigned. Values are adjusted modulo the range of the destination type when converted to a smaller width, and remain unchanged when converted to a larger width.

Operations. KIND 2 supports C-style arithmetic, logical, shift, and comparison operations over machine integers. Lustre's integer operators `+`, `-`, `*`, `div`, and `mod` are overloaded to apply also to two machine integers of the same type and return a machine integer of that type.

The integer comparison operators `>`, `<`, `>=`, `<=`, `=` are overloaded to the corresponding binary operations over machine integers of the same type. They all output a boolean value.

There are new machine integer operators for bit-wise conjunction (`&&`), disjunction (`||`), and negation (`!`), all with the expected arity and type, as well as left shift (`lsh`) and right shift (`rsh`) operators. The last two are both binary: the two inputs must have the same width but only the first can be signed. The output is signed if the first input is signed, and is unsigned otherwise; it is obtained by shifting the first input by the number of positions indicated by the second input. Right-shifting when the first operand is signed results in an arithmetic right shift, where the sign bit is preserved. A left-shift is equivalent to multiplication by 2 (modulo the width), and a right-shift is equivalent to division by 2. In other words, the left shift operator shifts towards the most-significant bit and the right shift operator shifts towards the least-significant bit.

To check safety properties of Lustre models with machine integers KIND 2 relies on off-the-shelf SMT solvers by leveraging their support for the theory of bit vectors of fixed width. Currently, only the SMT solvers Z3 [6] and cvc5 [1] support logics that allow the *combined* use of mathematical integers and machine integers. To use any of the other supported SMT solvers, the Lustre input must contain only boolean and machine integer types.

KIND 2's manual [7] provides more detailed information on machine integer support and on which SMT solvers are recommended for different combinations of data types in the input model.

In future work, we plan to extend KIND 2 to support floating point types as well.

3 Realizability Checking of Contracts

Contract-based software development is a major methodology for the rigorous construction of component-based reactive systems, embedded systems in particular. Contracts provide a mechanism for capturing the information needed to specify and reason about component-level properties at a desired level of abstraction. In this paradigm, a component C can be associated with a contract specifying its input-output behavior in terms of guarantees provided by C when its environment satisfies certain assumptions. Contracts are an effective way to establish boundaries between components and can be used to facilitate proofs of global properties of a complex system prior to its construction. Such proofs capitalize on the fact that complex components are typically specified simply as the composition of lower-level components. However, they are also built upon the implicit assumption that each leaf-level component contract in the system hierarchy is *realizable*. Roughly speaking, this means that it is possible to construct a component that, for any input allowed by the contract assumptions, can produce an output satisfying the contract guarantees. Unfortunately, without tool support it is all too easy for system designers to write leaf-level contracts that are unrealizable.

In KIND 2 the behavior of each component, or *node* in Lustre terminology, can be specified by providing either a set of equations that define the component's output in terms of its input and internal state (a *low-level* specification), or an assume-guarantee contract (a *high-level* specification), or both. The syntax restrictions and semantics of the Lustre language ensure that every low-level specification of a component is *executable* in the sense that for each possible input and internal state for the component there is a *unique* output and next state for the component to move to. Hence low-level specifications in Lustre are realizable by construction. When both specifications are provided in KIND 2, the low-level specification is expected to be a refinement of the high-level one. KIND 2 checks this by verifying that every execution that satisfies the former also satisfies the latter. Informally, we say that the set of equations *satisfy* the contract. In compositional reasoning, when only a contract is provided for a subcomponent C, KIND 2 assumes the existence of such a component when checking the properties of components that use C. This, however, may lead to bogus compositional proof arguments when C's contract is unrealizable.

KIND 2 now provides an option to check whether the contract of a component with no low-level specification is realizable. When a contract is unrealizable, the only way to fully explain why the contract is impossible to satisfy is to provide a *counter-strategy*, a (temporal) description of an *environment* for the component that prevents any potential realization of that component.

A user can examine a counter-strategy to try understand the reasons the contract is unrealizable, and fix it accordingly. However, as pointed out by Könighofer et al [13], a counter-strategy may be very large and complex, especially if it was generated automatically. For this reason, instead of a counter-strategy, Kind 2 provides examples of execution scenarios that lead to impossible conditions. Specifically, it outputs a single, finite computation path all of whose transitions satisfy the contract but whose end state has no outgoing transitions that satisfy the contract. In other words, Kind 2 provides concrete evidence for the existence of a *reachable deadlocking* state d for any putative realization of the contract. In addition, to facilitate the comprehension of the deadlocked state further, Kind 2 also provides a state d' such that transitioning from d to d' would minimize the number of violated contract guarantees. The (non-empty) set of the violated guarantees in question is returned as well.

When the contract is proven unrealizable, the user has also the option of invoking a sanity check on whether the contract is *satisfiable* at all, i.e., whether it is possible to construct a component such that for *at least* one input sequence allowed by the contract assumptions, there is some output value that the component can produce to satisfy the contract guarantees.

The realizability check implemented in Kind 2 is largely based on a synthesis procedure for infinite-state reactive systems, called JSyn-vg, by Katis et al. [12]. The main difference is that while the original work relies on a dedicated solver to implement the functionality provided by the AE-VAL procedure [8] of JSyn-vg, our implementation only requires a generic quantifier elimination procedure for the underlying data theories supported by Kind 2 (Booleans, linear integer arithmetic, and linear real arithmetic). Such quantifier elimination capabilities are provided by state-of-the-art SMT solvers such as Z3 [6] and cvc5 [1].

A detailed description of Kind 2's realizability checking functionality and an experimental evaluation comparing our implementation and the original implementation of JSyn-vg in the JKind model checker is available in a technical report [17].

4 Merit and Blame Assignment

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 in helping 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. For instance, Kind 2 can produce an independently checkable proof certificate for each of the properties it claims to hold [18]. Since these proof certificates are meant for automated proof checkers, however, they usually provide limited user-level insights on what elements of the system model contribute to the satisfaction of a property.

Kind 2 now offers two new diagnostic features that provide additional information on a chosen set of verified properties [15]: (*i*) the identification of minimal sets of model elements that are *sufficient* to prove the properties together with the subset of model elements that are *necessary* to prove those properties; (*ii*) the computation of minimal sets of model constraints whose violation leads the system to falsify one of more of the chosen properties.

Although these two pieces of information are closely related, they can be naturally mapped to difference typical use cases 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 human users in the synthesis of optimal implementations. With the latter, the goal is to determine the tolerance and resilience of a system against faults or adversarial environments due to natural causes 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 [5, 10]; impact analysis [19], design optimization; and robustness analysis [21, 22].

The merit assignment functionality relies on the concept of inductive validity core introduced by Ghassabani et al. [9]. Generally speaking, given a set of *model elements* M and an invariance 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 in contracts, node calls, equations in node definitions, and assertions[2]. Note that M itself is an IVC, although not a very interesting one. In practice, for complex enough models, smaller IVCs exist. In fact, it 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 the IVC to be *minimal*, that is, have no proper subsets that are also an 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 [19]: a set of *MUST* elements which are required for the satisfiability of P in every case, a set of *MAY* elements which are optional, and a set of elements that are irrelevant. This categorization

[2] Assertions are *unchecked* assumptions on a node's input. They are deprecated in Kind 2, in favor of contract assertions, but still supported for being part of Lustre.

provides complete traceability between specification and design elements, and can be used for coverage analysis [10] and for tracking the safety impact of model changes [15]. For instance, a change to an element e in the *MAY* set for P will not affect the satisfaction of P but will definitely impact some other property Q if e 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 [22] or resiliency to cyber-attacks [21]. For instance, an empty MUST set for a system S and one of its invariants P indicates that the property is satisfied by S in various alternative ways, making the system tolerant to faults or resilient against cyber-attacks as far as 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 P. Given a set of model elements M, 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.

We refer the interested reader to a related publication [15] and technical report [16] for implementation details and experimental results on merit and blame assignment with KIND 2.

References

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[2] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *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)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[3] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. 2016. CoCoSpec: A Mode-Aware Contract Language for Reactive Systems. In *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)*, Rocco De Nicola and eva Kühn (Eds.). Springer, 347–366. https://doi.org/10.1007/978-3-319-41591-8_24

[4] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The Kind 2 Model Checker. In *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)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 510–517. https://doi.org/10.1007/978-3-319-41540-6_29

[5] Hana Chockler, Daniel Kroening, and Mitra Purandare. 2010. Coverage in interpolation-based model checking. In *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, Sachin S. Sapatnekar (Ed.). ACM, 182–187. https://doi.org/10.1145/1837274.1837320

[6] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[7] Kind 2 Developers. 2022. *The Kind 2 user manual.* The University of Iowa. https://kind.cs.uiowa.edu/kind2_user_doc/

[8] Grigory Fedyukovich, Arie Gurfinkel, and Aarti Gupta. 2019. Lazy but Effective Functional Synthesis. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 92–113. https://doi.org/10.1007/978-3-030-11245-5_5

[9] Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. 2016. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 314–325. https://doi.org/10.1145/2950290.2950346

[10] Elaheh Ghassabani, Andrew Gacek, Michael W. Whalen, Mats Per Erik Heimdahl, and Lucas G. Wagner. 2017. Proof-based coverage metrics for formal verification. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 194–199. https://doi.org/10.1109/ASE.2017.8115632

[11] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. 1992. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE. *IEEE Trans. Software Eng.* 18, 9 (1992), 785–793. https://doi.org/10.1109/32.159839

[12] Andreas Katis, Grigory Fedyukovich, Huajun Guo, Andrew Gacek, John Backes, Arie Gurfinkel, and Michael W. Whalen. 2018. Validity-Guided Synthesis of Reactive Systems from Assume-Guarantee Contracts. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10806)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 176–193. https://doi.org/10.1007/978-3-319-89963-3_10

[13] Robert Könighofer, Georg Hofferek, and Roderick Bloem. 2013. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 563–583. https://doi.org/10.1007/s10009-011-0221-y

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

[15] Daniel Larraz, Mickaël Laurent, and Cesare Tinelli. 2021. Merit and Blame Assignment with Kind 2. In *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12863)*, Alberto Lluch-Lafuente and Anastasia Mavridou (Eds.). Springer, 212–220. https://doi.org/10.1007/978-3-030-85248-1_14

[16] Daniel Larraz, Mickaël Laurent, and Cesare Tinelli. 2021. Merit and Blame Assignment with Kind 2. *CoRR* abs/2105.06575 (2021). arXiv:2105.06575 https://arxiv.org/abs/2105.06575

[17] Daniel Larraz and Cesare Tinelli. 2022. Realizability Checking of Contracts with Kind 2. *CoRR* abs/2205.09082 (2022). https://doi.org/10.48550/arXiv.2205.09082 arXiv:2205.09082

[18] Alain Mebsout and Cesare Tinelli. 2016. Proof certificates for SMT-based model checkers for infinite-state systems. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 117–124. https://doi.org/10.1109/FMCAD.2016.7886669

[19] Anitha Murugesan, Michael W. Whalen, Elaheh Ghassabani, and Mats Per Erik Heimdahl. 2016. Complete Traceability for Requirements in Satisfaction Arguments. In *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*. IEEE Computer Society, 359–364. https://doi.org/10.1109/RE.2016.35

[20] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *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)*, Warren A. Hunt Jr. and Steven D. Johnson (Eds.). Springer, 108–125. https://doi.org/10.1007/3-540-40922-X_8

[21] Kit Siu, Abha Moitra, Meng Li, Michael Durling, Heber Herencia-Zapana, John Interrante, Baoluo Meng, Cesare Tinelli, Omar Chowdhury, Daniel Larraz, et al. 2019. Architectural and Behavioral Analysis for Cyber Security. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*. IEEE, 1–10.

[22] Danielle Stewart, Jing Janet Liu, Michael W Whalen, Darren Cofer, and Michael Peterson. 2020. Safety Annex for the Architecture Analysis and Design Language. (2020).