The KIND 2 Model Checker *

Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli

The University of Iowa

Abstract. KIND 2 is an open-source, multi-engine, 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 that allows the specification of assume-guarantee-style contracts for system components. KIND 2 was implemented from scratch based on techniques used by its predecessor, the PKIND model checker. This paper discusses a number of improvements over PKIND in terms of invariant generation. It also introduces two main features: contract-based compositional reasoning and certificate generation.

1 Introduction

KIND 2 is an SMT-based model checker for synchronous reactive systems. It relies on off-the-shelf SMT solvers to prove or disprove quantifier-free regular safety properties of models written in an extension of the synchronous dataflow language Lustre [11]. These properties can be expressed, in a separate annotation language, as invariants or as assume-guarantee-style contracts. KIND 2 is inspired by its predecessor PKIND [14] and uses several of the same techniques. However, it was engineered and implemented from scratch. Both checkers have several model checking engines, based on various techniques, which run concurrently and in cooperation, with the goal of proving or disproving properties and contracts.

KIND 2 is open-source and distributed in binary and source-code form under a liberal license at `http://kind.cs.uiowa.edu`. This paper focuses on its novel features, in particular, powerful invariant generation techniques, contract-based compositional reasoning, and proof certificate generation.

2 Functionality and Main Features

We start with a summary of KIND 2's basic functionality, *i.e.*, (dis)proving safety properties of reactive systems, and then describe KIND 2's distinguishing features.

Safety analysis. Lustre is a dataflow language that allows one to define system components as *nodes*, each of which maps a continuous flow of inputs (of various basic types) to continuous flows of outputs based on both current input values and previous input and output values (see Figure 1 for a simple example). Bigger components can be built by parallel composition of smaller ones, achieved syntactically with *node calls*. Through the use of observers [12], any (LTL) regular safety property can be expressed in Lustre as an invariant property, hence KIND 2 focuses on checking just invariant properties.

* The development of KIND 2 was partially funded by AFRL grant FA8750-13-C-0051, NASA grants NNA13AA21C, NNX14AI09G, and NNL14AA06C, and by Rockwell-Collins.

After various transformations and slicing, KIND 2 encodes Lustre nodes internally as state transition systems $\langle \mathbf{s}, I(\mathbf{s}), T(\mathbf{s}, \mathbf{s}') \rangle$ where $\mathbf{s}$ is the 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 $\mathbf{s}$). An *invariant property* P for such a system is a predicate over the variables $\mathbf{s}$ that must hold in every reachable state of the system. Instances of I, T and P are quantifier-free first-order formulas over the theories of equality with uninterpreted functions and linear integer and real arithmetic.

The node construct allows one to specify modular and hierarchical systems. KIND 2 takes advantage of this by performing *modular reasoning* over nodes. Each node can be assigned its own properties and verified individually. The results of the verification process (*e.g.*, proven properties and auxiliary invariants) can be reused in the analysis of other components calling that node. KIND 2 takes this approach further by allowing the user to specify assume-guarantee-style contracts for each node, effectively enabling *compositional reasoning* by fine-grained abstraction of sub-components.

```
node sofar (x:bool) returns (p:bool);
let p = (true -> pre p) and x; tel

node sum (x:int) returns (s:int);
let s = x + (0 -> pre s);
--! PROPERTY sofar(x > 0) => s > 0;  tel
```

Fig. 1: Example of annotated Lustre. Node `sofar` encodes the "always in the past" operator of pLTL.

At the component level, given an encoding $S \triangleq \langle \mathbf{s}, I(\mathbf{s}), T(\mathbf{s}, \mathbf{s}') \rangle$ of a Lustre node and a property P, KIND 2 tries to verify that P is invariant for S using a combination (described in Section 2) of different induction-based model checking engines: k-induction [16], IC3 [3] and various auxiliary invariant generation methods. K-induction is a generalization of standard induction and consists in finding a value k for which P holds in all reachable states within $k-1$ steps (base case), and is preserved by transition chains of length k (step case). IC3 is a popular directed reachability approach that iteratively strengthens the given property until it becomes inductive. We use an extension of IC3 to infinite-state systems which is based on an efficient form of approximate quantifier elimination. In our experience, IC3 is often complementary to k-induction as it can prove properties that are not k-inductive for any k while k-induction can handle properties that IC3 finds hard to strengthen to an inductive one. The invariant generation engines of KIND 2 produce on the fly auxiliary invariants that are used to incrementally strengthen the transition relation T, increasing the chances of proving the step case of k-induction and facilitating the job of IC3.

Incremental and modular invariant generation. PKIND introduced an invariant generation technique parameterized by a partial order $\preceq$ over some (equality) type τ [13]. It starts from a set of candidate terms $\mathbb{C}$ of type τ over a system S and heuristically produces invariants of the form $c \preceq c'$ and $c = c'$ where $c, c' \in \mathbb{C}$. For the `bool` type, used in Lustre both for Boolean state variables and for properties, $\preceq$ is implication and $\mathbb{C}$ is constructed by mining the initial state predicate and the transition relation of S for Boolean terms. The approach maintains an index k and a directed acyclic graph (DAG), whose vertices are sets of terms from a partition of $\mathbb{C}$. A vertex $V = \{c_1, c_2, \ldots, c_n\}$ denotes the chain of equalities $c_1 = c_2 = \cdots = c_n$. An edge from node V to V' denotes the inequality $c \preceq c'$ for any term c in V and c' in V'. The DAG is a compact representation of a set of invariant conjectures about S. Initially, $k = 0$ and the DAG has a single node $\mathbb{C}$, conjecturing that all the terms in $\mathbb{C}$ are equivalent in every reachable state of S. This conjecture is tested with a Bounded Model Checking-style query to an SMT solver for a counterexample k states away from an initial state. If none is found, the conjecture is correct for states reachable in up to k steps from an initial one, and k is incremented. Otherwise, the DAG is modified by removing edges or splitting nodes so that its refined conjecture is consistent with the latest counterexample and all previous ones. The algorithm refines its DAG and increments k until k reaches a user-specified upper bound d. It then performs a multi-property $(d+1)$-induction step check over each element of the conjecture. Any equality or inequality between two candidate terms in the conjecture that is i-inductive for $i \leq d$ will be proved and communicated as invariant.

We have modified this technique so that it progresses in *lockstep*. When the conjecture is correct at depth k, the invariant generation engine of KIND 2 performs the $(k+1)$-induction step check right away. This allows it to output invariants that are k-inductive for a small k faster. An additional benefit is that there is no need for a user-defined upper bound d, whose value can vastly influence runtimes—for instance on large systems, where unrolling the transition predicate several times can be extremely expensive.

Furthermore, KIND 2 can execute this invariant generation technique *modularly* when the input system is defined as the composition of two or more nodes. In that case, the subsystem hierarchy is traversed bottom-up. For each subsystem S, a set of $(k+1)$-inductive invariants (with k initially 0) is obtained as discussed above. Those invariants are then instantiated in every subsystem that has S as a direct subcomponent, recursively. Once the process reaches the top-level system, any invariants discovered at that level are communicated to the other reasoning engines of KIND 2. At that point a new bottom-up traversal starts with a greater value of k. This approach has two significant advantages with respect to running invariant generation on the full system monolithically: (i) it discovers invariants for subsystems more easily and quickly; (ii) it is self-reinforcing since instances of the invariants discovered for a subsystem of a component S can be used to help prove invariant conjectures for S.

Compositional reasoning. *Compositional reasoning* is a popular technique to improve the scalability of verification tools on systems defined as hierarchies of components.[1] Components have *contracts* enforcing their use in a certain context in order for them to guarantee certain properties (Figure 2 for an example). Analyzing a component consists in checking that its contract holds after abstracting at call-site all of its (possibly complex) sub-components by their own contract. A contract for a system $S \triangleq \langle \mathbf{s}, I(\mathbf{s}), T(\mathbf{s}, \mathbf{s}') \rangle$ is a pair $C \triangleq \langle A(\mathbf{s}), G(\mathbf{s}) \rangle$ where, informally, the *assumption* predicate A describes properties that S expects its inputs to have, while the *guarantee* predicate G expresses how the component behaves when A holds at all times. A contract can introduce local variables (streams), refer to previous values of streams, and call arbitrary Lustre nodes. This makes KIND 2's contract language

```
node max (x:real) returns (m:real);
let m = x -> if x > pre x then x
             else pre x; tel

node avg (x,y:real) returns (a:real);
(*@contract
   assume x <= y;
   guarantee x <= a and a <= y; *)
let a = (x + y) / 2.0; tel

node sav (x:real) returns (s:real);
(*@contract
   assume x > 0.0 and x > pre x;
   guarantee s <= max(x); *)
let s = avg(x -> pre s, x); tel
```

Fig. 2: Lustre nodes with contracts.

[1] For simplicity, we describe here only the case of asymmetric parallel composition, where there are no feedback loops between components, although KIND 2 can deal with the general case.

expressive enough to represent any regular safety properties, once they are recast in terms of past temporal logic (see [6] for more details on the contract language and its use). In KIND 2, verifying that S satisfies its contract reduces to verifying that $G(\mathbf{s})$ is an invariant for the system $S_A = \langle\, \mathbf{s},\, I(\mathbf{s}) \wedge A(\mathbf{s}),\, A(\mathbf{s}) \wedge T(\mathbf{s},\mathbf{s}') \wedge A(\mathbf{s}')\, \rangle$. If S is a component of some larger system S', which provides it with input values, then S can be abstracted by its guarantee G at call-site in S' as long as the assumption A at call-site is an invariant for S'. If it is, we say the call is *safe*. If the call is unsafe, then so is S' since it does not respect the contract of S. If all components of a system verify their contract and make only safe calls then the overall system is safe. KIND 2 can construct this argument via a *modular* analysis, where system components are analyzed bottom-up in the subsystem hierarchy with a process similar to modular invariant generation.

Refinement. KIND 2's modular and compositional analysis of multi-component systems resorts to contract refinement when needed. Consider a system S_1 with contract $C_1 = \langle A_1(\mathbf{s}_1), G_1(\mathbf{s}_1)\rangle$ that uses a subsystem S_2 with contract $C_2 = \langle A_2(\mathbf{s}_2), G_2(\mathbf{s}_2)\rangle$. Suppose that KIND 2 cannot prove S_1's contract compositionally, that is, by abstracting S_2 by its contract. A reason for this might be that the abstraction provided by C_2 is too weak. KIND 2 will then *refine* S_1 in the analysis by replacing S_2's contract with S_2 itself, provided, however, that the following conditions are met: *(i)* S_2 is safe (i.e., it verifies its contract and does not make unsafe calls), and *(ii)* all calls to S_2 in S_1 are provably safe. If the new analysis succeeds, the user is notified of the specific contract abstraction under which the result was obtained. Otherwise, the refinement process continues recursively until no more contract refinement is possible. When a system like S_2 is used instead of its contract C_2 it is because it provably admits a smaller set of execution traces than C_2. Because of this, when analyzing a newly refined system, KIND 2 retains any invariant/property already proved and any information on properties that are still unproven or falsified. This means that when the analysis restarts after refinement, KIND 2 will only check the proof obligations that were not previously discharged, in effect restarting precisely from where the previous analysis had stopped.

Certification. Having to trust the results of complex model checkers like KIND 2 is a source of concern for some users. To address this problem, KIND 2 can produce an independently checkable *proof certificate* for the properties that it claims to have proven for a (sub)system.[2] This certificate is in the form of a *k-inductive invariant* (expressed as a formula together with a specific value of k) that implies all the proven properties. This form is general enough that it can be effectively produced by all the model checking engines described previously. Certificates coming from these engines are combined conjunctively thanks to the fact that a k-inductive invariant is also k'-inductive for any $k' \geq k$. Individual certificates are initially generated by single engines based on their deductions regarding some set of properties and invariants. The combined certificate is then simplified along two dimensions, the value of k and the size and complexity of the invariant itself, using various fixpoint-based heuristics relying on unsat cores and counterexamples to induction. The final certificate output by KIND 2 is written in SMT-LIB 2 format and embedded in an SMT-LIB 2 script that checks that the certificate

[2] Currently, certificate generation is available only for monolithic analyses. An extension to compositional ones is planned as future work.

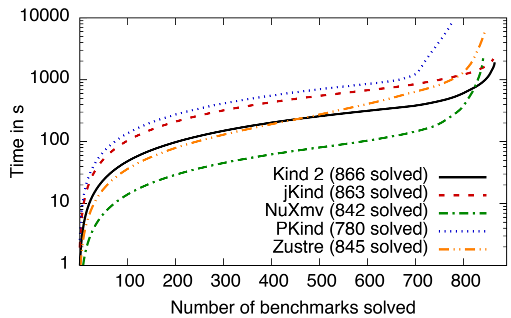

Fig. 3: Comparison between KIND 2 and other infinite-state model checkers.

Tool	k-induction	IC3
PKIND	yes+**ig**	no
ZUSTRE	no	yes+**i**
KIND 2	yes+**m**+**ig**	yes
JKIND	yes+**ig**	yes+**ia**
NUXMV	no	yes+**ia**

Table 1: Techniques implemented in the tools.

is k-inductive and implies the proven input properties. As a first approximation, any SMT-LIB 2-compliant solver can then be used as a *certificate checker*. This essentially shifts the burden of trust from KIND 2 to the SMT solver, reducing the trusted core to the latter. In our initial empirical evaluation, this approach allows Kind 2 to *generate* and *check* certificates, with an SMT solver, with a reasonable overhead (in all cases, less than an order of magnitude). We are currently working on eliminating the SMT solver as well from the trusted core by capitalizing on the proof-producing capabilities of certain SMT solvers. Specifically, in collaboration with the developers of the CVC4 solver [2], we are instrumenting Kind 2 to generate from CVC4 a final certificate in the LFSC language [17]. This way, the trusted core will reduce even further, to the much simpler LFSC proof checker.

Architecture. KIND 2 is written in OCaml and has a concurrent architecture similar to that of PKIND. Its various engines (base case and inductive step of k-induction, IC3, invariant generation, and so on) run simultaneously and in cooperation. They exchange information, mostly about properties proved or disproved to be invariant, through a message passing interface implemented on top of the ZeroMQ library. The concurrent execution of the base (BMC) of k-induction with the step case makes KIND 2 efficient at disproving properties. This architecture provides superior support for systems with multiple components and properties since it allows KIND 2 to check several properties per component at the same time and output counterexamples or proven properties incrementally, as it discovers them. Various off-the-shelf SMT solvers (currently, CVC4 [2], Yices [9], and Z3 [8]) are used as backend reasoning engines.

3 Experimental Evaluation

Compositionality and certificate generation make KIND 2's internal architecture more complex, and with a higher potential overhead, than comparable model checkers. So we provide an evaluation of KIND 2's performance as a monolithic model-checker first (without certificate generation), before discussing the performance of its compositional reasoning features.

Comparison with other tools. We compared KIND 2 with a number of recent model checkers for infinite-state systems: PKIND [14]; JKIND [10], a model checker similar to PKIND developed in Java by Rockwell-Collins; ZUSTRE, a Lustre front end for the Z3-based model checker Spacer [15]; and NUXMV [5], a general purpose model checker for synchronous finite-state and infinite-state systems. Table 1 shows the techniques implemented by each tool among a modular version (**m**) of *k*-induction with or without invariant generation (**ig**), and IC3 possibly augmented with interpolation (**i**) or implicit predicate abstraction [7] (**ia**). We ran each tool on a Linux machine with two 12-core 64-bit AMD Opteron processors and 32GB of memory on a set of single-property benchmarks that includes those discussed in [14].[3] NUXMV was given encoded versions of the original Lustre problems in its own input language, which were provided to use by its developers. We gave a timeout of five minutes for each problem. Figure 3 shows that KIND 2 is very competitive with its peers, outperforming its predecessor PKIND and providing an answer (either *valid* or a counterexample) in more cases than any other tool.

Compositional vs. monolithic verification. We evaluated compositional reasoning in KIND 2 on the TCM (Transport Class Model) for medium-sized aircraft discussed and verified compositionally *by hand* by Brat *el al*. [4]. The subsystem of the TCM we had access to, which is modeled in Lustre, includes components for the latitudinal and longitudinal controllers, and for the mode logic that decides which controller should be active at any time. The controllers are heavily numerical and contain non-linear expressions, which are problematic for current SMT solvers. We wrote contracts corresponding to Federal Aviations Regulations [4] for most of the components of the subsystem. We also abstracted non-linear expressions by components with a linear contract.

The runtime to verify *every component* of the system bottom-up, *including* the abstractions of non-linear expressions, is about 400 seconds on a 2014 i7 CPU running OSX. A comparison with a purely monolithic approach is not possible because of the presence of non-linearity. All SMT solvers we tried would return *unknown*, even for checks dealing with a single, relatively simple component. As a consequence, we did a monolithic analysis of a modified TCM system where the non-linear expressions are replaced by their linear contract but otherwise nothing else is abstracted. In this setting, the analysis of the top level of the system ran for two hours without reaching a conclusion. We refer the interested reader to Champion *el al*. [6] for a more in-depth discussion.

4 Applications

KIND 2 is used in academia and in a variety of industrial settings. For the latter, it is for instance one of the backend model checkers in the AGREE framework for compositional verification of AADL models [1] at Rockwell-Collins. It has been used at General Electric for model-based test case generation. It is also used in an open-source model-checking plugin for Simulink developed by NASA Ames and CMU, which relies on Lustre model checkers and produces user feedback at the Simulink block level. KIND 2's proof certificates are leveraged as an innovative way to approach tool qualification with respect to DO-178C requirements in a NASA and FAA funded project.

[3] The set is publicly available at `https://github.com/kind2-mc/kind2-benchmarks`.

References

1. J. Backes, D. D. Cofer, S. P. Miller, and M. W. Whalen. Requirements analysis of a quad-redundant flight control system. In *NASA Formal Methods - 7th Int'l Symposium, NFM 2015*, volume 9058 of *LNCS*, pages 82–96. Springer, 2015.
2. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification - 23rd Int'l Conference, CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
3. A. R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th Int'l Conference, VMCAI 2011*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
4. G. Brat, D. H. Bushnell, M. Davies, D. Giannakopoulou, F. Howar, and T. Kahsai. Verifying the safety of a flight-critical system. In *Formal Methods - 20th Int'l Symposium, FM 2015*, volume 9109 of *LNCS*, pages 308–324. Springer, 2015.
5. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In *Computer Aided Verification - 26th Int'l Conference, CAV 2014*, volume 8559 of *LNCS*, pages 334–342. Springer, 2014.
6. A. Champion, A. Gurfinkel, T. Kahsai, and C. Tinelli. CoCoSpec: A mode-aware contract language for reactive systems. In *Software Engineering and Formal Methods, 14th Int'l Conference, SEFM 2016*, LNCS. Springer, 2016. (To appear).
7. A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th Int'l Conference, TACAS 2014*, volume 8413 of *LNCS*, pages 46–61. Springer, 2014.
8. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th Int'l Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
9. B. Dutertre. Yices 2.2. In *Computer Aided Verification - 26th Int'l Conference, CAV 2014*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.
10. A. Gacek, A. Katis, M. W. Whalen, J. Backes, and D. D. Cofer. Towards realizability checking of contracts using theories. In *NASA Formal Methods - 7th Int'l Symposium, NFM 2015*, volume 9058 of *LNCS*, pages 173–187. Springer, 2015.
11. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
12. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology, AMAST 1993*, Workshops in Computing, pages 83–96. Springer, 1993.
13. T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *NASA Formal Methods - Third Int'l Symposium, NFM 2011*, volume 6617 of *LNCS*, pages 192–206. Springer, 2011.
14. T. Kahsai and C. Tinelli. Pkind: A parallel k-induction based model checker. In *Proceedings 10th Int'l Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011*, volume 72 of *EPTCS*, pages 55–62, 2011.
15. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *Computer Aided Verification - 26th Int'l Conference, CAV 2014*, volume 8559 of *LNCS*, pages 17–34. Springer, 2014.
16. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design, Third Int'l Conference, FMCAD 2000*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.
17. A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 41(1):91–118, Feb. 2013.