The JKind Model Checker

Andrew Gacek\textsuperscript{1}, John Backes\textsuperscript{1}, Mike Whalen\textsuperscript{2}, Lucas Wagner\textsuperscript{1}, and Elaheh Ghassabani\textsuperscript{2}

\textsuperscript{1} Rockwell Collins  
\textsuperscript{2} University of Minnesota

\textbf{Abstract.} J\textsc{Kind} is an open-source industrial model checker developed by Rockwell Collins and the University of Minnesota. J\textsc{Kind} uses multiple parallel engines to prove or falsify safety properties of infinite state models. It is performance competitive with other state-of-the-art model checkers and serves as the back-end for various industrial applications.

1 Introduction

J\textsc{Kind} is an open-source\textsuperscript{3} industrial infinite-state inductive model checker for safety properties. Models and properties in J\textsc{Kind} are specified in L\textsc{ustre} \cite{7}, a synchronous data-flow language, using the theories of linear real and integer arithmetic. J\textsc{Kind} uses SMT-solvers to prove and falsify multiple properties in parallel.

A distinguishing characteristic of J\textsc{Kind} is its focus on the quality of results. For a proven property, J\textsc{Kind} provides traceability between the property and individual model elements. For a falsified property, J\textsc{Kind} provides options for simplifying and generalizing the counterexample in order to highlight the root cause of the failure. In industrial applications, we have found this additional quality of the results to be at least as important as the primary results.

Another important characteristic of J\textsc{Kind} is that it is designed to be integrated directly into user-facing applications. Written in Java, J\textsc{Kind} runs on all major platforms and is easily compiled into other Java applications. J\textsc{Kind} bundles the Java-based S\textsc{MT}I\textsc{nterpol} solver and has no external dependencies. However, it can optionally call Z\textsc{3}, Y\textsc{ices 1}, Y\textsc{ices 2}, C\textsc{vc4}, and M\textsc{athsat} if they are available.

2 Functionality and Main Features

J\textsc{Kind} is structured as several parallel engines that coordinate to prove properties, mimicking the design of K\textsc{ind} \cite{9}. Some engines are directly responsible for proving properties, others aid that effort by generating invariants, and still others are reserved for post-processing of proof or counterexample results. Each engine can be enabled or disabled separately based on the user’s needs. The core

\footnote{https://github.com/agacek/jkind}
architecture of the solving engines in JKind is shown in Figure 1. Although displayed as point-to-point connections, information between engines is actually broadcast so any engine may pick the information it wants.

The post-processing engines are the greatest difference between JKind and more traditional model checkers. In industrial application, the quality of an answer is just as important as the answer itself. For example, more than knowing a property is true, we want to know why it is true. We want to trace the property back to the model and see which parts were exercised. This type of coverage and traceability information is often required for certification of safety critical systems [10].

Each engine of JKind is described below.

**Bounded Model Checking (BMC).** The BMC engine performs a standard iterative unrolling of the transition relation to find counterexamples and to serve as the base case of \( k \)-induction. The BMC engine guarantees that any counterexample it finds is minimal in length.

**\( k \)-induction.** The \( k \)-induction engine performs the inductive step of \( k \)-induction, possibly using invariants generated by other engines.

**Invariant Generation.** The invariant generation engine uses a template-based invariant generation technique [8] using its own \( k \)-induction loop.

**Property Directed Reachability (PDR).** The PDR engine performs property directed reachability [4] using the implicit abstraction technique [2]. Unlike BMC and \( k \)-induction, each property is handled separately by a different PDR sub-engine. Invariants generated as a side-product of PDR are shared with the \( k \)-induction process.

**Inductive Validity Cores (IVC).** For a proven property, an inductive validity core is a subset of LUSTRE equations from the input model for which the property
still holds [6]. It indicates which part of the model is relevant to the proof of the property. The IVC engine uses a heuristic algorithm to efficiently produce nearly minimal inductive validity cores. As a side-product, the IVC algorithm also minimizes the set of invariants used to prove a property and reports back this reduced set.

**Advice.** The advice engine saves and re-uses results between runs of JKind, improving performance. When enabled, a set of invariant from a previous run of JKind, so called advice, is read in and (hopefully) re-verified. Because advice is re-verified, it does not need to be completely accurate. In particular, advice from a previous run of JKind can be used on a model even if that model is further modified. Pieces of advice that successfully re-verify are used as invariants while those that do not are discarded.

**Smoothing.** Counterexamples generated from BMC and PDR are not always easily understood, so as an optional post-processing step we smooth counterexamples to minimize the number of changes to the input variables. The smoothing engine uses a MAXSAT query over the original BMC-style unrolling of the transition relation combined with weighted assertions that each input variable does not change on each step. The MAXSAT query therefore tries (as best as possible) to hold all inputs constant while still falsifying the original property. This engine is only available with SMT-solvers that support MAXSAT such as Yices and Z3.

**Interval Generalization.** Another post-processing step available for counterexamples is to perform interval generalization where individual values in a counterexample are replaced by intervals that still falsify the property. This analysis is done purely through simulation, without using an SMT-solver. It is useful in showing that, for example, the value of a variable on a given step only needs to be positive and its particular value does not matter.

### 3 Integration & Applications

JKind is the back-end for a variety of user-facing applications. In this section, we briefly highlight a few and how they employ the features discussed previously.

We wrote JKind in Java which makes it multi-platform and very easy to integrate into other Java applications. Moreover, we created the JKindAPI package which contains utilities for creating LUSTRE specifications, calling JKind, processing JKind results, graphically displaying real-time results, and nicely formatting counterexamples. Many of the applications in this section make heavy use of JKindAPI.

**Specification and Analysis of Requirements**

The Specification and Analysis of Requirements tool, SPEAR for short, is a prototyping and analysis tool for requirements expressed in formal notations [5]. SPEAR captures requirements in a way that is backed by the formal semantics of LUSTRE, which enables them to be analyzed using model checking to ensure they are correct and consistent.
**SPEAR** uses **JKind** to prove properties over requirements, and uses IVC to create a traceability matrix between requirements and properties. This quickly highlights unused requirements, over-constrained properties, and other common problems. **SPEAR** also uses **JKind** for test case generation using the Unique First Cause criteria [12] by creating *trap properties*. Each trap property is expected to be falsifiable, but in such a way that the counterexample has exactly the desired properties for a given test case. **SPEAR** uses smoothing in **JKind** to ensure the resulting test cases are simple and understandable.

**Assume Guarantee Reasoning Environment**  
*Open Source*

**JKind** is used as the default model checker for the Assume Guarantee Reasoning Environment (AGREE) [3]. **AGREE** refers to both an embedded language annex in the Architectural Analysis and Design Language (AADL) and to a plugin for the OSATE AADL Integrated Development Environment. The **AGREE** annex annotates the AADL model with formal requirements, and the plugin reasons about these requirements. The purpose of **AGREE** is to model behavioral requirements of an embedded system using formal assume guarantee contracts. The plugin generates Lustre specifications that are checked by **JKind**.

**AGREE** makes use of multiple **JKind** features including smoothing and interval generalization to present clear counterexamples, IVC to show requirements traceability, and counterexample generation to check the consistency of an AADL component’s contract. **AGREE** also uses **JKind** to generate various test cases for component contracts. The test case generator uses counterexamples (or lack thereof) to demonstrate requirements coverage.

**Flight Control Logic**  
*Proprietary*

A flight control group at Rockwell Collins develops state machines which manage flight modes of an aircraft. These state machines often have few states, but many possible transitions between states. There is a priority ordering to resolve conflicts between transitions, but this can make it hard to fully exercise the behavior of a state machine. The flight controls group has developed a particular graphical representation to manage this complexity.

We have developed a tool which formalizes these state machine models. It translates the state machine to **Lustre** and uses **JKind** to check properties over the state machine. We use IVCs to establish traceability, i.e. which transitions are covered by which properties. We also use trap properties to generate test cases exercising transitions. Here IVCs are used when a trap property is true which means it failed to generate a test case. The IVC indicates which transitions are blocking the transition we wish to exercise in the test case. The tool presents all this information graphically to its users.

**MC/DC Test Case Generation**  
*Proprietary*

The Engine-Indicating and Crew-Alerting System (EICAS) group at Rockwell Collins develops equations which determine which messages and alerts should be delivered to pilots. Certification for this system requires extensive test cases which must satisfy criteria such as the Modified Condition/Decision Coverage (MC/DC) metric. We have developed a tool which generates these tests by trans-
lating the EICAS equations into Lustre and generating MC/DC trap properties in JKind. Smoothing is very important in this context as test cases need to be run on the actual hardware where timing is not precisely controllable. Thus, test cases with a minimum of changes to the inputs are ideal.

**Model-based Fuzzing**

As part of an ongoing DARPA program, Rockwell Collins is developing a tool for generating fuzz tests from system models. This tool takes a Lustre description of a system and uses JKind to generate test cases which exercise different portions of the model. Effective fuzz testing requires a huge number of test cases which is not feasible to generate solely with a model checker. Instead, this fuzzing tool uses interval generalization in JKind to generate generalized counterexamples which it then randomly samples at high speed. In fact, research in this area has recently expanded interval generalization to trapezoidal generalization which is strictly more general, but still allows very fast random sampling.

**Static IMPerative AnaLyzer**

The Static IMPerative AnaLyzer (SIMPAL) is a tool for performing compositional reasoning over software [11]. SIMPAL features a specification language, called Limp, for modeling programs that utilize preexisting components. Limp is a Lustre-like imperative language. It provides control flow elements, global variables, and a syntax for specifying preconditions, postconditions, and global variable interactions of preexisting components. SIMPAL translates Limp programs to an equivalent Lustre representation which is passed to the JKind to perform assume-guarantee reasoning, reachability, and viability analyses. The feedback from these analyses is used to refine the program to ensure that the software functions as intended.

4 Discussion

**Related Work** JKind is one of a number of similar infinite-state inductive model checkers including Kind 2, Kind, NuXmv, and Zustre. These tools each offer multi-engine solvers that utilize both k-induction and some variant of IC3/PDR. In a recent comparison [1], JKind was the second most capable solver in terms of the number of problems solved (behind Kind 2) and had competitive performance across a large benchmark suite. The most noticeable bottleneck in JKind is the start-up time for the Java Virtual Machine (JVM). This cost is insignificant for larger models but causes decreased performance for benchmarks consisting of many very small models.

**Conclusion** JKind is similar to a number of other solvers that each solve infinite state sequential analysis problems. Nevertheless, it has some important features that distinguish it. First, a focus on quality of feedback to users for both valid properties (using IVCs) and invalid properties (using smoothing and counterexample generalization). Second, it is supported across all major platforms and is straightforward to port due to its implementation in Java. Third, it
is small, modular, and well-architected, allowing straightforward extension with new engines. Fourth, it is open source with a liberal distribution license (BSD), so it can be adapted for various purposes, as demonstrated by the number of tools that have incorporated it.

Acknowledgments
The work presented here was sponsored by DARPA as part of the HACMS program under contract FA8750-12-9-0179.

References
1. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (2016)
2. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Grenoble, France, April 5-13, 2014. Proceedings. pp. 46–61 (2014)
3. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Proceedings of the 4th International Conference on NASA Formal Methods, pp. 126–140. NFM'12 (2012)
4. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD ’11, Austin, TX, USA, October 30 - November 2, 2011. pp. 125–134 (2011)
5. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: Spear v2.0: Formalized past LTL specification and analysis of requirements. In: NASA Formal Methods – 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings (2017)
6. Ghassabani, E., Gacek, A., Whalen, M.W.: 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. pp. 314–325 (2016)
7. Halbwachs, N., Caspi, P., Raymond, P., Piliaud, D.: The synchronous dataflow programming language lustre. Proceedings of the IEEE 79(9), 1305–1320 (September 1991)
8. Kahsai, T., Garoche, P.L., Tinelli, C., Whalen, M.: Incremental verification with mode variable invariants in state machines. In: Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012). vol. 7226, pp. 388–402 (April 2012)
9. Kahsai, T., Tinelli, C.: Pkind: A parallel k-induction based model checker. In: Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011. pp. 55–62 (2011)
10. RTCA DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Washington, DC (December 2011)
11. Wagner, L., Greve, D., Gacek, A.: SIMPAL: A compositional reasoning framework for imperative programs. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (2017)
12. Whalen, M.W., Rajan, A., Heimdahl, M.P., Miller, S.P.: Coverage metrics for requirements-based testing. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis. pp. 25–36. ISSTA ’06 (2006)