PKIND: A parallel k-induction based model checker[*]

Temesghen Kahsai
The University of Iowa
temesghen-kahsaiazene@uiowa.edu

Cesare Tinelli
The University of Iowa
cesare-tinelli@uiowa.edu

PKIND is a novel parallel *k*-induction-based model checker of invariant properties for finite- or infinite-state Lustre programs. Its architecture, which is strictly message-based, is designed to minimize synchronization delays and easily accommodate the incorporation of incremental invariant generators to enhance basic *k*-induction. We describe PKIND's functionality and main features, and present experimental evidence that PKIND significantly speeds up the verification of safety properties and, due to incremental invariant generation, also considerably increases the number of provable ones.

1 Introduction

PKIND is a parallel model checker based on the *k*-induction principle, used to verify invariant properties of programs written in the specification/programming language Lustre [8]. 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). PKIND assumes an idealized version of Lustre, which treats int as the type of mathematical integers, and real as the type of rational numbers. Its reasoning about Lustre programs is done in the context of a first-order quantifier-free logic that includes uninterpreted functions and mixed real-integer linear arithmetic. Idealized Lustre programs can be faithfully and readily encoded as transition systems in this logic (see [7] for more details). PKIND relies on the SMT solvers CVC3 [3] and Yices [5], in alternative, as satisfiability solvers for this logic.

PKIND's architecture is strictly message-based and designed to minimize synchronization delays and easily accommodate the concurrent automatic generation of invariants to bolster basic *k*-induction. A first level of parallelism is introduced in the *k*-induction procedure itself by executing the base and the inductive steps concurrently. A second level allows the addition of one or more independent processes that incrementally generate *auxiliary* invariants for the system being verified. These invariants are fed to the *k*-induction loop as soon as they are produced and used to strengthen the induction hypothesis.

To the best of our knowledge, this sort of parallel architecture has not been presented in previous work on parallel model checking. Our approach is orthogonal to those in previous work [2] that focus on other sources of parallelism, including parallelization across the processes of an asynchronous transition system. Most closely related to ours is the work by Eén *et al.* [6] who describe a sequential implementation of SAT-based *k*-induction in which a Bounded Model Checking loop is interleaved with one performing just the inductive step of *k*-induction. Our approach goes beyond that work, not only in using a genuinely parallel architecture, but also by incorporating concurrent invariant generation processes. Another line of related work is exemplified by [1, 13, 4], which discuss a different type of parallelism in the BMC algorithm. There, satisfiability checks are done concurrently within the SAT solver. This too is orthogonal to our approach, as the parallelism we exploit is not at the level of the underlying solver, but at the level of the *k*-induction procedure.

[*]Work partially supported by AFOSR grant #FA9550-09-1-0517 and NSF grant #1049674.

Jiri Barnat and Keijo Heljanko (Eds.)
10th International Workshop on
Parallel and Distributed Methods in verifiCation (PDMC 2011)
EPTCS 72, 2011, pp. 55–62, doi:10.4204/EPTCS.72.6

In the current version of PKIND, invariant generation is achieved using a novel incremental version of an offline invariant discovery scheme we developed in previous work [9]. This general scheme consists in sifting through a large set of formulas generated automatically from a transition system's description, looking for possible invariants. The formulas in the set, the *candidate invariants*, are all instances of a template encoding a decidable relation over the system's data types. In [9], a single invariant is generated at the end of the process as a conjunction of template instances, each of which is k-inductive for some k. In contrast, in the version developed for PKIND, instances that are k-inductive for the same k are discovered and returned before instances that are k'-inductive for some $k' > k$.

Before describing PKIND's architecture, we briefly recall the definition of k-induction [12]. Assume a logic $\mathscr{L}$ and a transition system S specified in the logic by an initial state condition $I(\mathbf{x})$ and a two-state transition relation $T(\mathbf{x},\mathbf{x}')$ where $\mathbf{x},\mathbf{x}'$ are vectors of state variables. A state property $P(\mathbf{x})$ is invariant for S, i.e., satisfied by every reachable state of S, if the following entailments hold in $\mathscr{L}$ for some $k \geq 0$:

$$I(\mathbf{x_0}) \wedge T(\mathbf{x_0},\mathbf{x_1}) \wedge \cdots \wedge T(\mathbf{x_{k-1}},\mathbf{x_k}) \models P(\mathbf{x_0}) \wedge \cdots \wedge P(\mathbf{x_k}) \tag{1}$$

$$T(\mathbf{x_0},\mathbf{x_1}) \wedge \cdots \wedge T(\mathbf{x_k},\mathbf{x_{k+1}}) \wedge P(\mathbf{x_0}) \wedge \cdots \wedge P(\mathbf{x_k}) \models P(\mathbf{x_{k+1}}) \tag{2}$$

A counterexample trace for the base case entailment (1) indicates that the property P is falsified in a reachable state of S, and so is not invariant. A counterexample trace for the induction step entailment (2) does not provide the same information because it may start from an unreachable state. The normal way to try to rule out such spurious counterexamples is to increase the depth k of the induction. This is, however, not guaranteed to succeed because some invariant properties are not k-inductive for any k. An additional, and orthogonal, way of enhancing k-induction is to strengthen the induction hypothesis $P(\mathbf{x_0}) \wedge \cdots \wedge P(\mathbf{x_k})$ with the assertion of previously proven invariants for S.[1]

2 PKind's architecture

PKIND is implemented in OCaml and is built with components from (the sequential) KIND model checker [7] and the KIND-INV invariant generator [9] we developed in previous work. Its concurrent components are implemented as operating system processes since OCaml's concurrency model does not take advantage of multi-processor hardware to parallelize thread-level computation. We used the MPI (*Message Passing Interface*) API [11] to implement communication among the different processes.

Figure 1 illustrates the general architecture of PKIND. For simplicity, we consider only the case of one invariant generation process. The extension to an arbitrary number of invariant generators (of the same type) is straightforward. Each process uses its own copy of the SMT-solver; and each of them receives as input the formulas encoding the Lustre program to be checked. The three processes, described below, cooperate as in the Actor model, exchanging messages asynchronously by means of non-blocking send and receive operations on message queues.

Base process: Starting from $k = 0$, the base process checks the entailment of case (1) in the k-induction definition, for increasing values of k. If the entailment fails, it produces a counterexample and sends a termination signal M_2 to the inductive step process. Otherwise, it keeps increasing k and checking the entailment until it receives a message M_1 from the inductive step process stating that the process has proven the inductive step (2) for a certain value n of k. At that point, the base case process checks that the base case holds for that value of k, succeeding if it does and returning a counterexample otherwise.

[1] Further improvements involve the addition of *path compression* constraints and checks [12, 10], which not only speed up computation but also guarantee completeness in certain cases—including all finite state systems.

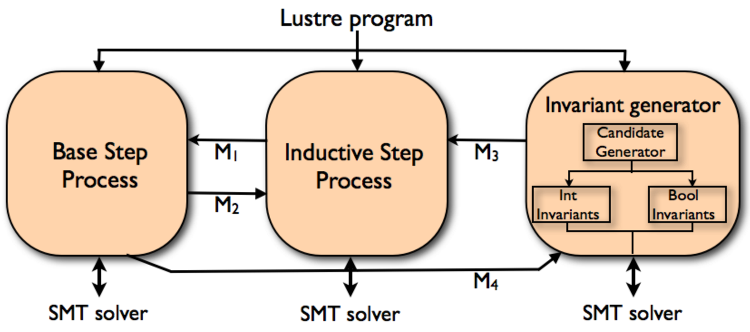

Figure 1: PKIND general architecture.

Inductive step process: This process checks the inductive step entailment for increasing values of k until one of the following occurs, in this order: (i) the entailment succeeds, (ii) it receives a termination signal from the base case process, or (iii) it receives an invariant from the invariant generation process. In the latter case, the inductive step process asserts the discovered invariant for all the states involved so far (from 0 to $k+1$) before repeating the process with the same value of k. The inductive step process has an auxiliary role in this architecture—it is the base case process that determines whether the property is k-inductive or not.

Invariant generation process: This process is composed of three modules. The *Candidate generator*, synthesizes a set of candidate invariant from predefined templates. The *Int* and *Bool Invariants* modules generate integer and Boolean type invariants respectively. Invariants are sent to the inductive step process, in messages M_3, as soon as they are discovered. The process keeps sending newly discovered invariants until it processes all candidates or it receives a termination signal M_4 from the base process.

This process produces k-inductive invariants for a given transition system $\mathscr{S}$ from a *template* $R[\_,\_]$, a formula of $\mathscr{L}$ representing a decidable binary relation over one of system's data types. The invariants are conjunctions of instances $R[s,t]$ of the template produced with terms s,t from a set U of terms over the state variables of $\mathscr{S}$. The set U can be determined heuristically from $\mathscr{S}$ (and possibly also from the property P to be proven) in any number of ways.[2]

2.1 Invariant Generation

The general invariant generation scheme introduced in [9] is illustrated in Figure 2, Version A. The two-phase procedure, also based on k-induction, starts with a conjunction C of candidate invariants and first eliminates from C as many conjuncts as possible that have an actual counterexample. Then it attempts to prove the resulting C k-inductive, pruning any unprovable conjunct until it succeeds—possibly with an empty C in the worst case. All conjuncts of C that remain at the end are invariant and can be returned. Note that in Phase 1 the search for counterexamples stops when, for some k, C is falsified by no k-reachable states. This is just a heuristic termination condition which does not preclude the possibility that some conjuncts of C may be falsified by longer counterexamples. As a consequence, every conjunct of C that does not pass the test in Phase 2 are conservatively assumed not to be invariant (even if it may be k'-inductive for some $k' > k$) and removed.

[2] In our experiments, we constructed U with terms occurring in the $\mathscr{L}$-encoding of $\mathscr{S}$ plus a few distinguished constants from the domain of $\mathscr{S}$'s variables.

```
proc invariant_generator ≡
  $C = \bigwedge_{s,t \in U} R[s,t]$
  $k = -1$
  $\mathsf{reset}_{\mathrm{base}}$; $\mathsf{reset}_{\mathrm{ind}}$
  // Phase 1
  repeat
    $k = k+1$
    $\mathsf{assert}_{\mathrm{base}}(T_k)$
    $changed = \mathsf{false}$
    while $\neg\,\mathsf{entailed}_{\mathrm{base}}(C_k)$ do
        $cex = \mathsf{project}(\mathsf{cex}_{\mathrm{base}}, C_k)$
        $C = \mathsf{filter}(C, cex)$
        $changed = \mathsf{true}$
  until $\neg\, changed$
  // Phase 2
  $\mathsf{assert}_{\mathrm{ind}}(T_1 \wedge C_0 \wedge \cdots \wedge T_{k+1} \wedge C_k)$
  while $\neg\,\mathsf{entailed}_{\mathrm{ind}}(C_{k+1})$ do
      $cex = \mathsf{project}(\mathsf{cex}_{\mathrm{ind}}, C_{k+1})$
      $C = \mathsf{filter}(C, cex)$
  $\mathsf{send}(D, \mathsf{ind\_proc})$
```

Version A
(non-incremental)

```
proc invariant_generator ≡
  $C = \bigwedge_{s,t \in U} R[s,t]$
  $k = -1$
  $\mathsf{reset}_{\mathrm{base}}$
  repeat
    $k = k+1$
    $\mathsf{assert}_{\mathrm{base}}(T_k)$
    while $\neg\,\mathsf{entailed}_{\mathrm{base}}(C_k)$ do
        $cex = \mathsf{project}(\mathsf{cex}_{\mathrm{base}}, C_k)$
        $C = \mathsf{filter}(C, cex)$
    $D = C$
    $\mathsf{reset}_{\mathrm{ind}}$
    $\mathsf{assert}_{\mathrm{ind}}(T_1 \wedge D_0 \wedge \cdots \wedge T_{k+1} \wedge D_k)$
    $changed = \mathsf{false}$
    while $\neg\,\mathsf{entailed}_{\mathrm{ind}}(D_{k+1})$ do
        $cex = \mathsf{project}(\mathsf{cex}_{\mathrm{ind}}, D_{k+1})$
        $D = \mathsf{filter}(D, cex)$
        $changed := \mathsf{true}$
    $\mathsf{send}(D, \mathsf{ind\_proc})$
  until $\neg\, changed$
```

Version B
(incremental)

Figure 2: **General scheme for invariant generation.** The procedures reset and assert and the functions entailed and cex, are all indexed by the copy of the $\mathscr{L}$-solver used by the process. reset empties the solver's set of asserted formulas; $\mathsf{assert}(F)$ adds the formula F to that set; $\mathsf{entailed}(F)$ returns true iff the currently asserted formulas $\mathscr{L}$-entail F. When invoked after a call to entailed that returned false, cex returns a counterexample for that failed entailment. The function project takes an assignment α for state variables and a formula F, and returns the restriction of α to the variables of F. The function filter takes a conjunctive property P and an assignment α for P's variables, and returns the property obtained from P by removing all conjuncts falsified by α. In Version B, D is meant to be a copy of C, not a renaming. The call $\mathsf{send}(D, \mathsf{ind\_proc})$ sends D to the inductive step process.

The general invariant discovery scheme above produces a single conjunctive invariant at the very end. In a concurrent setting, however, it is better for runtime performance to have an incremental invariant generation process, which identifies and returns invariants as it goes. Concretely, it is better for the invariant generation process of PKIND first to identify, and immediately send to the inductive step process, conjuncts of C that are 0-inductive, then identify and send those that are 1-inductive, and so on.

Pseudo-code for the incremental procedure is provided in Figure 2, Version B. The procedure starts again with the conjecture $C = \bigwedge_{s,t \in U} R[s,t]$. However, it does the following for every $k \geq 0$. First, it eliminates from C all conjuncts falsified by a k-reachable state. Then it makes a copy D of C and tries to prove it k-inductive by checking the k-induction step on D. Counterexamples in that step are used to weaken D further, eliminating more and more conjuncts from it until no counterexamples exists. The final formula D is k-inductive and can be already sent to the inductive step process. If D did not need to be weakened at all, it means that *all* the conjuncts left in C after the base case check were k-inductive. Hence, the whole process terminates. If D was weakened, the eliminated conjuncts are either falsified by a longer counterexample or possibly k-inductive for a larger k. Those conjuncts are still in C, so in that case the procedure increases k by 1 and repeats with the larger k.

We point out that the conjuncts left in D (and sent out because k-inductive) are not removed from C before repeating. This is for simplicity but also for convenience because on the new iteration they will end up be in D again, strengthening the induction hypothesis for the inductive step check. That means, however, that the set of conjuncts sent out at iteration $k+1$ is a (strict) superset of the one sent out at iteration k. This can be remedied by keeping the previous D and sending out only the difference between the new D and the old.[3]

2.2 Additional Features

PKIND takes full advantage of all the sophisticated features of its two embedded SMT solvers, such as their being on-line, incremental, backtrackable, and able to compute and return satisfying assignements or unsatisfiable cores of input formulas. It provides three different running modes: in `k_induct` mode, PKIND functions as a basic parallel k-induction model checker, scheduling only the processes for the base and the inductive step; in `no_inc_invariant` mode, it creates also the invariant generation process. The invariant generation implemented here is Version A of Figure 2; in `inc_invariant` mode, the invariant generation is done in an incremental fashion, as in Version B of Figure 2.

PKIND includes a number of optional enhancements on top of this basic procedure, inherited from the KIND checker. The main ones are: *path compression*, *termination checking* and *abstraction*. Path compression strengthens the induction hypothesis with a formula that, in essence, removes from consideration program executions with repeated states or more than one initial state. Termination checking allows it to prove a non-k-inductive property in some cases by realizing that the reachable state space has been completely explored. Abstraction generates a structural abstraction of the idealized Lustre program and performs something similar to a CEGAR loop in both the base case and the induction step, with the goal of improving the tool's scalability. See [7] for more information on these and other minor enhancements.

3 Experimental evaluation

We have evaluated PKIND and its different working modes experimentally against KIND and KIND-INV using the same benchmark set as in [9].[4] That set collects a variety of benchmarks from several sources, with each benchmark consisting of a Lustre program and a property to be checked for invariance.

Let us call a benchmark *valid* if its property holds in every reachable state of the associated program, and *invalid* otherwise. KIND is able to show 438 of the 941 benchmarks in our set invalid by returning a (separately verified) counter-example trace for the program. KIND reports 309 of the remaining benchmarks as valid and diverges on the remaining 194 benchmarks, even with very large timeout values.

For the experiments described here the benchmark set is divided in two groups: the 503 valid or unsolved benchmarks, and the 438 invalid ones. Our tests compare PKIND vs. KIND along two dimensions: *precision*, measured as the percentage of solved benchmarks, and *runtime*. For the latter, we were interested specifically in evaluating how the parallel architecture speeds up k-induction in the case of invalid benchmarks, and how *incremental* invariant generation contributes in decreasing solving times for valid benchmarks. The experiments were run on a 6-core 2.67 GHz Intel Xeon machine, with 6 GB of physical memory, under RedHat Enterprise Linux 4.0. Version 1.0.9 of the Yices solver was used both for PKIND and KIND.

[3] In our experimental evaluation, however, this enhancement did not seem necessary because in most cases the first few sets of invariants were enough for the process ind_proc to prove the input property.

[4] Tools and experimental data can be found at `http://clc.cs.uiowa.edu/Kind`.

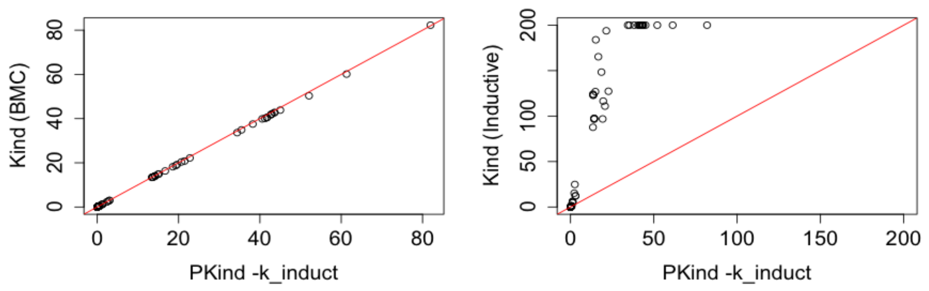

Figure 3: A comparison of generating counter-examples between PKIND `-k_induct` and KIND on invalid benchmarks. On the left-hand side we compare PKIND `-k_induct` and KIND in *BMC* mode. On the right-hand side we compare PKIND `-k_induct` and KIND in *inductive* mode.

3.1 Precision results

As in [9], we used KIND-INV to generate invariants offline. We first ran KIND-INV on the benchmark set for 200 seconds. For each of the benchmarks where KIND-INV did not time out, we obtained a set of invariants, and added them to the program as a single conjunctive assertion. The asserted formula was the constant true when KIND-INV timed out or ended up discarding all conjectures from the initial set. Then, we ran KIND in inductive mode with a 100s timeout on each invariant-enhanced benchmark. KIND's precision over the 503 valid or unsolved benchmarks grows from 61%, without invariants, to 85%, with the added invariants. For each additional benchmark proved valid, the property goes from (most likely) not k-inductive for any k to k-inductive for $k \leq 16$.

Using the same time out value, in `k_induct` mode PKIND proves valid the same benchmarks proven by KIND with no invariants. In either `no_inc_invariant` and `inc_invariant` mode, PKIND proves the same benchmarks proven by KIND with invariants, despite the fact that it generates invariants on the fly, whereas KIND gets them as input.[5] Precision is also unchanged when comparing PKIND with KIND run (in a much faster) BMC mode on the set of 438 invalid benchmarks. Furthermore, in all these cases, solving times are shortened considerably with PKIND, as discussed below.

3.2 Runtime results

The two scatter plots in Figure 3 compare the runtimes of PKIND `-k_induct` with those of KIND respectively in BMC mode (left-hand side plot), which performs just bounded model checking, and inductive mode (right-hand side plot), which performs full k-induction. The timeout for both tools was 200s. As the first plot clearly shows, the runtimes of the two tools are pretty much the same in the first case. Due to some overhead of the MPI implementation in PKIND, for some benchmarks KIND is slight faster. The average solving time for PKIND `-k_induct` is 2.34s, while for KIND in *BMC* mode it is 2.29s. The superiority of PKIND is clearly illustrated with the second plot. For a more quantitative comparison, both tools are able to produce a counterexample in less than 1s per benchmark for 90% of all invalid benchmarks. However, PKIND `-k_induct` takes less than 82s to solve any of the invalid benchmarks, whereas KIND in inductive mode times out on half of them. It is worth stressing that PKIND `-k_induct` is also superior from a usability perspective because it combines the speed of KIND's BMC mode over invalid inputs with the generality and precision of KIND's inductive mode, without requiring the user to chose between these two modes.

[5] As a precaution against implementation errors, each invariant produced by KIND-INV or by PKIND was independently and successfully verified by giving it to KIND separately as a property to prove for the program from which it was generated.

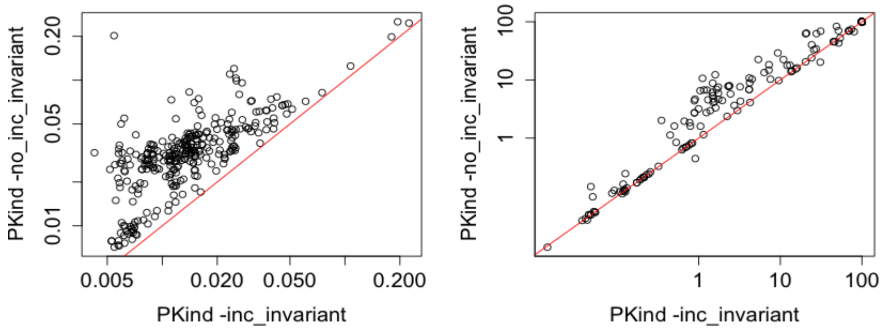

Figure 4: A comparison of solving the benchmarks between PKind `-no_inc_invariant` and PKind `inc_invariant` on valid benchmarks. On the left-hand side we compare the two modes on benchmarks which are valid without the addition of invariants. On the right-hand side we compare the two tools on the benchmarks previously unsolved. Both plots use a log-log scale over seconds.

Moving to PKind's invariant generating modes, since invariants for Kind are generated offline, it is not possible to do direct runtime comparisons between the two systems over the valid or unsolved benchmarks. A more interesting comparison is between the two invariant generating modes themselves. In a nutshell, our results show that the addition of incrementally produced invariants, makes those benchmarks considerably faster to solve. In more detail, we first compared PKind `-no_inc_invariant` and PKind `-inc_invariant` on all previously valid benchmarks, those that can be shown valid without the addition of invariants. On those benchmarks, the conjunctive invariant produced by the non-incremental version has no effect on the k-induction loop, because the property is invariably proven before the invariant is generated. In contrast, when invariants are generated incrementally, they are in many cases sent to the inductive step process early enough to play a role in reducing the overall solving time.

The left-hand side of Figure 4 provides a comparison between the two invariant generation modes. The plot demonstrates how incremental invariant generation speeds up runtimes. Specifically, the incremental version is able to solve 76% of all the valid benchmarks in less than 0.02 seconds, while the non-incremental version can do that for only 17% of those benchmarks. Looking at the previously unsolved benchmarks, with a timeout of 100s, both invariant generation modes solve 66% of them. Again, however, the incremental version is faster than the non-incremental one although the speed up is less pronounced, as can be seen in the right-hand side plot of Figure 4. In more detail, the incremental version is able to solve 79% of all newly solved benchmarks in less than 9s, whereas the non-incremental version is able to solve 73% of them. It is interesting to observe that in a few cases non-incremental invariant generation is slight faster than the incremental one. The reason for this is that the useful invariant is one that is k-inductive for a relatively larger k than usual.[6] In those cases, incrementally sending and processing (useless) k-inductive invariants for lower values of k produces a bit of overhead.

With regard to the size of our benchmarks, the invalid ones range from 200 bytes to 40KB of Lustre source code with no comments and standard use of white space. The average time to solve an invalid benchmark of size more than 20KB is 9.12 seconds. For the valid benchmarks the size ranges from 400 bytes to 58KB. The average time to solve a valid benchmark of size more than 20KB bytes is 17.6 seconds. Although benchmark sizes may appear small, we stress that the majority of these benchmarks are infinite-state systems written in a high level specification language, so a benchmark's size is not a

[6] In most benchmarks in our set, adding 0- or 1-inductive invariants is enough to make the property provable.

good predictor of its difficulty. In fact, looking at runtime data over the whole benchmark set we found no correlation between benchmark size and solving times.

4 Conclusion and future work

We have described PKIND, a novel parallel k-induction-based model checker. PKIND is designed to keep interprocess synchronization overheads to a minimum and easily accommodate the concurrent automatic generation of invariants to bolster the basic k-induction procedure. While any independent invariant generation techniques could be used in principle in our architecture, we have developed a new one, based on our previous work, that produces invariants in an incremental fashion to exploit the advantages of the parallel setting. Our experimental results provide an initial and clear evidence that the architecture significantly speeds up the verification of safety properties. In addition, due to the incremental invariant generation, this architecture considerably increases the number of provable benchmarks.

In future work, we plan to study new ways of automatically generating invariants for k-induction, both in parallel, i.e., using abstract interpretation techniques, and on-demand, in response to counterexamples found by the inductive step. We are also working on a new version of PKIND built from scratch around the architecture discussed here and designed to accept other input languages besides Lustre.

References

[1] Erika Ábrahám, Tobias Schubert, Bernd Becker, Martin Fränzle & Christian Herde (2011): *Parallel SAT solving in bounded model checking* 21(1), pp. 5–21, doi:10.1093/logcom/exp002.

[2] J. Barnat, L. Brim, M. Češka & P. Ročkai (2010): *DiVinE: Parallel Distributed Model Checker (Tool paper)*. In: *HiBi/PDMC 2010*, IEEE, doi:10.1109/PDMC-HiBi.2010.9.

[3] Clark Barrett & Cesare Tinelli (2007): *CVC3*. In W. Damm & H. Hermanns, editors: *CAV'07*, LNCS, Springer, doi:10.1007/978-3-540-73368-3_34.

[4] Aaron R. Bradley (2011): *SAT-based model checking without unrolling*. VMCAI '11, Springer-Verlag, pp. 70–87. Available at `http://dl.acm.org/citation.cfm?id=1946284.1946291`.

[5] Bruno Dutertre & Leonardo de Moura (2006): *The YICES SMT Solver*. Technical Report, SRI International. Available at `http://yices.csl.sri.com/`.

[6] Niklas Eén & Niklas Sörensson (2003): *Temporal induction by incremental SAT solving*. *Electr. Notes Theor. Comput. Sci.* 89(4), doi:10.1016/S1571-0661(05)82542-3.

[7] George Hagen & Cesare Tinelli (2008): *Scaling up the formal verification of Lustre programs with SMT-based techniques*. In: *FMCAD '08*, Piscataway, NJ, USA, pp. 1–9, doi:10.1109/FMCAD.2008.ECP.19.

[8] N. Halbwachs, P. Caspi, P. Raymond & D. Pilaud (1991): *The synchronous data-flow programming language Lustre*. *Proceedings of the IEEE* 79(9), pp. 1305–1320, doi:10.1109/5.97300.

[9] Temesghen Kahsai, Yeting Ge & Cesare Tinelli (2011): *Instantiation-Based Invariant Discovery*. In: *NFM 2011*, *LNCS* 6617, Springer, pp. 192–207. Available at `http://dl.acm.org/citation.cfm?id=1986308.1986326`.

[10] Leonardo de Moura, Harald Rueß & Maria Sorea (2003): *Bounded Model Checking and Induction: From Refutation to Verification*. In: *CAV 2003*, *LNCS* 2725, Springer, doi:10.1007/978-3-540-70952-7_21.

[11] Peter S. Pacheco (1997): *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc.

[12] Mary Sheeran, Satnam Singh & Gunnar Stålmarck (2000): *Checking Safety Properties Using Induction and a SAT-Solver*. In: *FMCAD '00*, Springer-Verlag, London, UK, pp. 108–125, doi:10.1007/3-540-40922-X_8.

[13] Siert Wieringa, Matti Niemenmaa & Keijo Heljanko (2009): *Tarmo: A Framework for Parallelized Bounded Model Checking*. In: *PDMC '09*, pp. 62–76, doi:10.4204/EPTCS.14.5.