Scaling up the formal verification of Lustre programs with SMT-based techniques

George Hagen
Department of Computer Science
The University of Iowa, USA
ghagen@cs.uiowa.edu

Cesare Tinelli
Department of Computer Science
The University of Iowa, USA
tinelli@cs.uiowa.edu

October 8, 2008

Abstract

We present a general approach for verifying safety properties of Lustre programs automatically. Key aspects of the approach are the choice of an expressive first-order logic in which Lustre's semantics is modeled very naturally, the tailoring to this logic of SAT-based k-induction and abstraction techniques, and the use of SMT solvers to reason efficiently in this logic. We discuss initial experimental results showing that our implementation of the approach is highly competitive with existing verification solutions for Lustre.

1 Introduction

Lustre is a synchronous dataflow language commonly used in a number of manufacturing industries to model or implement reactive embedded systems [16]. Because of its declarative nature and simplicity, it is well suited to model checking and verification, in particular with regards to safety properties [17]. Existing formal methods tools for Lustre, however, do not seem to have kept pace with the latest developments in software model checking and verification, and are somewhat lacking in scope and scale. Currently, those can be obtained only by a translation into the specification languages of other tools—such as NuSMV, SAL, Prover, PVS, ACL2, and so on.

This work aims at bridging the current gap. Its main ingredient is the choice of an expressive logic with efficient automated reasoners for modeling Lustre programs and proving safety properties for them. Instead of temporal

reasoning based on *propositional* logic we rely on temporal reasoning based on a first-order logic with built-in theories for equality over uninterpreted symbols and linear mixed real and integer arithmetic.

An approximation of Lustre's denotational semantics can be modeled simply and naturally in this logic, with two distinctive advantages: *i*) infinite-state as well as finite-state Lustre programs can be encoded uniformly and *compactly* in the logic; *ii*) reasoning about these programs can be done directly in terms of Lustre's own dataflow model. This contrast with other approaches which are either mostly limited to finite-state programs or need a translation into some other specification language.

The idea of adopting logics of this sort in software model checking is not new [10, 1, 14, e.g.], and has been applied to the verification of relatively simple Lustre programs already in [13]. Recent advances in *Satisfiability Modulo Theories (SMT)* [4], however, make it now possible to apply this idea to a much wider range of real-world Lustre applications. Our use of SMT technology builds on previous research in SAT-based temporal induction (aka, k-induction) [21, 5], extended and adapted to the SMT case as in [10, 12], as well as on research in predicate abstraction and refinement.

Our main contribution is the combination of a selection of these techniques and their tailoring to Lustre. In particular, we extend SMT-based k-induction to incorporate abstraction and refinement, something that to our knowledge has not been reported before. The main outcome of our work is a new verification system that is highly competitive with current automated solutions for the verification of safety properties of Lustre programs.

In the following, we first sketch our approach, focusing on major aspects such as path compression and abstraction refinement. We then discuss an initial experimental evaluation of our implementation against current alternatives. This evaluation suggests that our new system considerably advances the state of the art, as it greatly outperforms current Lustre-specific verifiers. The study also shows that a direct approach like ours compares very favorably to translating Lustre programs into the language of a premier SMT-based model checker, SAL 3.0 [9], and using that checker.

Related Work. Our work is closest to Franzén's [13] which extends SAT-based k-induction to produce a safety property verifier for Lustre programs with unbounded integers. There, the extension is achieved by adding to the MiniSat SAT solver some simple ILP procedures for handling integer constraints. The focus of [13] was building the extended SAT solver. In contrast, we rely on much more powerful off-the-shelf embeddable SMT solvers that can also handle linear rational arithmetic constraints, and work more on improving the k-induction procedure.

Our path compression method of invariant strengthening and termination checking can be seen as a special case of the one described by de Moura *et al.* [10]. That work, however, also describes invariant strengthening techniques based on quantifier elimination, which we do not use.

A dependency-based abstraction method analogous to ours is described by

Chan *et al.* [6] for the dataflow language RSML. There, however, abstractions appear to be generated statically as a preprocessing step, and not refined dynamically. Our abstraction method is more similar to the one developed by Babic and Hu [2], which uses a SAT solver as its reasoning engine. The main idea of our method however—treating local variables as input ones and refining based on spurious counterexamples—goes back to Kurhshan [19] and appears in various forms in several works on hardware verification. Our use of unsatisfiable cores in our refinement heuristic recalls a similar approach by Chauhan *et al.* [7] for SAT-based model checking.

2 Preliminaries

Lustre is a declarative programming language for manipulating data flows, or *streams*, infinite sequences $(v_0, v_1, \ldots)$ of values of conventional data types, such as (bounded) integers, floating point numbers, and Booleans.

In essence, a Lustre function, called *node* in Lustre, is the declarative specification of a stream transformer, mapping a finite set of streams to another finite set of streams. Operationally, a node has a cyclic behavior: at each cycle i, it takes as input the value of each input stream at position or *instant*, i and returns the value of each output stream at instant i. Lustre nodes have a limited form of memory in that, when computing the output values they can also look at input and output values from previous instants, up to a finite limit statically determined by the program itself.

Figure 1 contains a simple example of a two-node Lustre program. The program models a thermostat whose target temperature, initially set to 70 degrees, can be changed by 1 degree at a time by pressing an "up" or a "down" button.

Typically, the body of a Lustre node consists in a set of *definitions*, equations of the form $x = t$ (as seen in the nodes in Figure 1) where x is a variable denoting an output or a locally defined stream and t is an expression, in a certain stream algebra, whose variables name input, output, or local streams. More generally, x can be a tuple of stream variables and t an expression evaluating to a tuple of the same type (as in the last equation of the node `therm_control` in Figure 1).

Most of Lustre's operators are pointwise liftings to streams of the usual operators over stream values. For example, let $x = (m_0, m_1, \ldots)$ and $y = (n_0, n_1, \ldots)$ be two integer streams. Then, $x + y$ denotes the stream $(m_0 + n_0, m_1 + n_1, \ldots)$; a constant like 5, say, denotes the constant integer stream $(5, 5, \ldots)$. Two important additional operators are a unary shift-right operator pre ("previous"), and a binary initialization operator $\rightarrow$ ("followed by"). The first is defined as $\mathsf{pre}(x) = (v, m_0, m_1, \ldots)$ with the value v left unspecified. The second is defined as $x \rightarrow y = (m_0, n_1, n_2, \ldots)$.

In Figure 1, the definition of the real-valued stream `target` constrains it to have value 70 at instant 0. At each instant $i > 0$ the current value of `target` is its previous value (at instant $i - 1$) minus 1 if the current value of the Boolean stream `down` is `true`; it is its previous value plus 1 if the current value of `up` is `true`; it is the previous value otherwise.

```
node thermostat (actual_temp, target_temp, margin: real)
 returns (cool, heat: bool) ;
let
 cool = (actual_temp - target_temp) > margin ;
 heat = (actual_temp - target_temp) < -margin ;
tel

node therm_control (actual: real; up, down: bool)
 returns (heat, cool: bool) ;
var target, margin: real ;
let
 margin = 1.5 ;
 target = 70.0 -> if down then (pre target) - 1.0
                  else if up then (pre target) + 1.0
                  else pre target ;
 (cool, heat) = thermostat (actual, target, margin) ;
tel
```

Figure 1: A simple Lustre program.

The right-hand side of an equation defining a stream can contain non-recursive calls to other nodes. Syntactical restrictions on the equations in a Lustre program guarantee that its streams are well defined.

2.1 From Lustre to SMT

To reason about the values that the streams of a Lustre program have at a given instant it would be convenient to have a logic in which Lustre's basic data types and their operations are built-in. Automated reasoning in such a logic, however, is not (yet) feasible. In alternative, we can consider an *idealized* version of Lustre with unbounded integers instead of machine integers and infinite-precision rationals instead of floating point numbers. Idealized Lustre programs can be modeled faithfully in a typed first-order logic with uninterpreted function symbols and built-in integers and rationals[1]. We will call it *Idealized Lustre Logic* and denote by $\mathcal{IL}$. The main lure of $\mathcal{IL}$ is that the validity in it of quantifier-free formulas with *linear* numerical terms is not only decidable but decidable rather efficiently by means of SMT techniques [4]. The main limitations are the exclusion of programs containing non-linear operations, and the inability to model behavior due overflows, underflows and rounding errors.

In this work, we adopted $\mathcal{IL}$ as the underlying logic. Future work will focus on extending our approach efficiently to programs with non-linear expressions and to a logic modeling real Lustre more accurately.

2.2 A logical model of Lustre

In $\mathcal{IL}$, a stream x of values of type τ can be represented simply as an uninterpreted function of type $\mathbb{N} \to \tau$. Then, equations between streams can be reduced to universally quantified equations between stream values by a straightforward

[1]Having tuples as well is convenient but not necessary because they can be treated as syntactic sugar.

homomorphic translation. For instance, the integer stream equation $x = y + z$ can be translated into $\forall n{:}\mathbb{N}.\, x(n) = y(n) + z(n)$, where x, y, and z become functions symbols of type $\mathbb{N} \to \tau$.[2] Similarly, the equation $x = y \to y + \mathsf{pre}\ z$ can be translated into $\forall n{:}\mathbb{N}.\, x(n) = ite(n = 0,\, y(0),\, y(n) + z(n-1))$ where ite is the if-then-else operator in the logic.

Let N be any single-node (idealized) Lustre program with stream variables $\mathbf{x} = \langle x_1, \ldots, x_p, y_1, \ldots, y_q \rangle$ where $x_1, \ldots, x_p$ are input variables and $y_1, \ldots, y_q$ are non-input (i.e., local and output) variables. The semantics of N is completely captured in $\mathcal{IL}$ by the universal quantification over the variable n of the following system of equations

$$\Delta(n) = \left\{ \begin{array}{rcl} y_1(n) & = & t_1[\mathbf{x}(n), \mathbf{x}(n-1), \ldots, \mathbf{x}(n-d)] \\ \vdots & \vdots & \vdots \\ y_q(n) & = & t_q[\mathbf{x}(n), \mathbf{x}(n-1), \ldots, \mathbf{x}(n-d)] \end{array} \right.$$

where each t_i is the translation of the expression defining y_i in N. Multi-node programs can be modeled the same way by first recursively in-lining in the main node all calls to subnodes. For the program in Figure 1, $\Delta(n)$ would look like the following (with m abbreviating `margin`, and so on):

$$\left\{ \begin{array}{l} m(n) = 1.5 \\ t(n) = ite(n = 0, 70.0, ite(d(n), t(n-1) - 1.0, \ldots)) \\ c(n) = (a(n) - t(n)) > m(n) \\ h(n) = (a(n) - t(n)) < -m(n) \end{array} \right.$$

We call an *(instantaneous) configuration (for N)* any well-typed tuple of values for $\mathbf{x}(n)$. Then, the program N can be understood as a system of constraints over instantaneous configurations. The notation for $\Delta(n)$ is suggestive of the fact that the value of $y_i(n)$, the $(p+i)$-th component of the configuration at instant n, is a function of (some of the values in) the configurations at instants $n, n-1, \ldots, n-d$. The value d is the maximum "nesting depth" of the pre operator in the body of N. For simplicity, from now on *we will restrict our presentation to programs with $d \leq 1$*. The extension to the general case, implemented in our verifier, is fairly straightforward.

A *trace* for the program N is a tuple $\mathbf{s} = \langle s_1, \ldots, s_{p+q} \rangle$ of streams of the same type as $\mathbf{x}$. A *path* (of length l) is a finite sequence (of l) consecutive configurations on a trace. The possible input/output behaviors of N are exactly the traces $\mathbf{s}$ that satisfy the formula $\forall n{:}\mathbb{N}.\, \Delta(n)$ when $\mathbf{x}$ is interpreted as $\mathbf{s}$. We call such traces a *legal traces*. A configuration is *reachable* if it occurs in some legal trace; it is *initial* if it is the first configuration on a legal trace. A path is *initial* if it is the initial segment of a legal trace.

As argued in [17], with Lustre programs one is mostly interested in verifying *safety properties*. Being able to checking *liveness properties* is not so crucial because Lustre is typically used to model real-time systems. In such systems,

[2] To simplify the notation we use the same metasymbol to denote both a Lustre variable and the corresponding function symbol in the translation.

progress must occur within predefined temporal limits and so it expressible as a safety property. More precisely, we are interested in properties invariant in the following sense.

Definition 1 *A property P of configurations is invariant (for N) if it holds for all reachable configurations.*

More generally, one can consider properties of paths, not of single configurations. We do not do that here, just for simplicity. We will consider only *quantifier-free properties*, properties expressible by a quantifier-free formula of $\mathcal{IL}$. While this an actual restriction, invariant properties checked in common practice are indeed quantifier-free.

Checking invariant properties of Lustre programs can be done by lifting and adapting to the logic $\mathcal{IL}$ a number of SAT-based model checking techniques. The main ones used in this work is k-induction.

3 SMT-based k-Induction

Let N be again a single node Lustre program, and let $\Delta(n)$ be the equational system modeling N in $\mathcal{IL}$. Let P be a property of N's configurations expressible by a quantifier-free formula $P(n)$ of $\mathcal{IL}$ over $\mathbf{x}(n)$. If t is any integer term of $\mathcal{IL}$, we denote by Δ_t the formula obtained from $\Delta(n)$ by replacing every occurrence of n with t. Similarly, for P_t.

3.1 Basic Procedure

Similarly to previous work on k-induction-based verification of transition systems [21, 5, 11, 10], we can prove that P is invariant for N if we succeed in proving the validity of the following two statements for some concrete $k \geq 0$:

$$\Delta_0 \wedge \Delta_1 \wedge \cdots \wedge \Delta_k \;\models_{\mathcal{IL}}\; P_0 \wedge P_1 \wedge \cdots \wedge P_k \tag{1}$$

$$\Delta_{\mathsf{n}} \wedge \Delta_{\mathsf{n}+1} \wedge \cdots \wedge \Delta_{\mathsf{n}+(k+1)} \wedge P_{\mathsf{n}} \wedge P_{\mathsf{n}+1} \wedge \cdots \wedge P_{\mathsf{n}+k} \;\models_{\mathcal{IL}}\; P_{\mathsf{n}+(k+1)} \tag{2}$$

where $\models_{\mathcal{IL}}$ denotes logical entailment in $\mathcal{IL}$ and n is an uninterpreted integer constant.

Both entailments can be decided by current SMT solvers for $\mathcal{IL}$. To verify P then, one asks the solver to prove both cases for some initial choice of k, retrying with a larger k until either the base case (1) is proven invalid or both the base case and the induction step (2) are proven valid. In the first situation, P is not invariant for N, and a counterexample path can be generated from an $\mathcal{IL}$-model of $\Delta_0 \wedge \cdots \wedge \Delta_k \wedge \neg(P_0 \wedge \cdots \wedge P_k)$ provided that the SMT solver is able to return models. In the second situation, P has been shown to hold for a set of configurations including all reachable configurations, which implies it is invariant.

This procedure is *sound*; it will never mistake a variant property for an invariant one. Standard arguments [12] can be used to show that in general

the procedure is not—and cannot be made—*complete* for general Lustre programs; specifically, the procedure may keep increasing k indefinitely for some invariant properties. Nevertheless, a number of improvements are possible to increase the procedure's *accuracy*, the set of invariant properties it can prove. Further improvements can also be applied to accelerate convergence to an answer. Two main enhancements to the basic k-induction procedure that we have found effective in practice are described in the next two sections.

The procedure's efficiency is also increased by exploiting several features of SMT solvers based on the *lazy approach* [4]. Similarly to previous SAT-based k-induction work [11, e.g.], we take full advantage of the fact that such solvers are on-line, incremental and backtrackable, return models, compute unsatisfiable cores, and can remember learned lemmas. Details on this and on other useful enhancements based on program slicing and other static preprocessing of the formulas in (1) and (2) can be found in [15].

3.2 Path Compression

A major enhancement of the basic procedure is represented by *path compression*, first introduced for k-induction in [5, 21]. In our setting, path compression is achieved by strengthening the left hand side of (1) and (2) so as to eliminate from consideration paths that contain repeated configurations or, more generally, configurations that are equivalent in an appropriate sense. A rather general notion of path compression for SMT-based k-induction is presented in [10]. Performance considerations usually prevent one from using that notion in its full generality. We use the one defined below, which seems better suited to the current capabilities of SMT solvers for $\mathcal{IL}$.

Let us assume for convenience that the argument of each application of the operator pre in the program N is just a stream variable.[3] We call those variables the *state variables* of N. Given a configuration $\mathbf{v}$ for N, the *state of* $\mathbf{v}$ is the subtuple of $\mathbf{v}$ corresponding to N's state variables. A path is *compressed* if no two configurations in it have the same state and none of them, except possibly the first, are initial.

We strengthen the premise of (2) by adding a quantifier-free formula $C_{\mathsf{n},k}$ over N's state variables that is satisfied by all and only those traces whose configurations from position n to $\mathsf{n}+k$ form a compressed path:

$$\begin{array}{l} \Delta_{\mathsf{n}} \wedge \Delta_{\mathsf{n}+1} \wedge \cdots \wedge \Delta_{\mathsf{n}+(k+1)} \wedge \\ P_{\mathsf{n}} \wedge P_{\mathsf{n}+1} \wedge \cdots \wedge P_{\mathsf{n}+k} \wedge C_{\mathsf{n},k} \end{array} \quad \models_{\mathcal{IL}} \quad P_{\mathsf{n}+(k+1)} \qquad (2')$$

It is immediate that using (2') instead of (2) preserves the k-induction procedure's accuracy. More interestingly, it also preserves its soundness, as proved in [15].

In addition to facilitating the SMT solver's task of proving $P_{\mathsf{n}+(k+1)}$, the restriction to compressed paths also yields a *complete* k-induction procedure

[3] We can always reduce ourselves to this case by first applying simple observational-equivalence-preserving transformations to N.

whenever the length of compressed initial paths is (statically) bounded above. Completeness then is achieved, while preserving soundness, by checking (1) and (2') for consecutive values of k starting at 0, and verifying before repeating the whole loop with a larger k whether there are any compressed initial paths of length $k+1$. In turn, this is done by checking that the entailment

$$\Delta_0 \wedge \cdots \wedge \Delta_k \quad \models_{\mathcal{IL}} \quad \neg C_{0,k+1} \tag{3}$$

holds. This is analogous to a loop check in Bounded Model Checking: if the test succeeds, no counterexamples will be found from $k+1$ on—because if they existed they could be compressed to counterexamples of length $k' < k+1$, against the fact that (1) held for k'. Hence, one can stop the induction loop and conclude that P is invariant for N.

3.3 Structural Abstraction

Abstraction is often helpful in scaling up the verification of properties of transition systems. One tries to prove a safety property for a given system S by proving it for a conservative abstraction $S^\sharp$ of S that is easier to deal with. This process may be followed by some refinement of the abstraction when the property does not hold for it. The same principle applies to Lustre programs as well, with some added advantages provided by Lustre's declarative nature.

Traditionally, *predicate abstraction*, a popular abstraction approach in software model checking [3], takes an infinite-state system S and a set of *abstraction predicates*, and generates a conservative (typically finite-state) abstraction of S. A main challenge in predicate abstraction is coming up with a good set of abstraction predicates.

With Lustre, a predicate completely capturing the behavior of a single node program N is readly provided by the equational system (predicate) Δ introduced in Section 2.1. This suggests a conceptually and practically much simpler yet effective form of abstraction, which we call *structural abstraction* following [2], based on the syntactical structure of N. In our case, it consists simply in removing equations from Δ. Conversely, refinement is achieved by adding back equations from Δ to the current abstraction $\Delta^\sharp \subseteq \Delta$.

At the Lustre level, this is essentially a form of *localization abstraction* [19] as it corresponds to turning some non-input streams of N into input ones, and removing their defining equation from the program. Clearly, the set of legal traces for the resulting program $N^\sharp$ contains all the legal traces for N. Therefore, every property invariant for $N^\sharp$ is also invariant for N. A property P that is invariant for N but not for $N^\sharp$ will have *spurious counterexamples*, legal traces for $N^\sharp$ that falsify P but are not legal traces of N.

We integrated the following counterexample-driven refinement strategy into our k-induction procedure. Instead of instances of the original system Δ, we use instances of its current abstraction $\Delta^\sharp$, initially consisting of the definitions of the variables of N that occur in the property to be checked. Whenever the base step (1) or the induction step (2) produces a spurious counterexample, we

refine $\Delta^\sharp$ as explained below and redo the step, repeating this process until no spurious conter-examples are returned for that step.

Note that we refine the predicate $\Delta^\sharp$ itself, not its single instances $\Delta^\sharp_0, \ldots, \Delta^\sharp_k$ or $\Delta^\sharp_{\mathsf{n}}, \ldots, \Delta^\sharp_{\mathsf{n}+(k+1)}$ used respectively in (1) and (2). Doing the latter is usually too fined grained, producing rather slow convergence.

We observe that it is possible to combine abstraction with path compression and loop detection. However, the combination must be done carefully since it can create subtle soundness issues (again, see [15]).

The Refinement Step

We experimented with several choices for refining $\Delta^\sharp$ and we settled on the following one, which seems to be the most effective in practice. To start, observe that Δ induces a (possibly cyclic) directed graph G over its terms, where for each equation $y = t$ in Δ, y is linked to t and t is linked to every stream variable occurring in it.

Refinement is done essentially by starting from a non-input variable y, and doing a depth-first traversal of G—taking care of breaking the loops in the graph—until some input variable x is reached. The set of all equations along the current path in G from y to x that are not already in $\Delta^\sharp$ are then added to $\Delta^\sharp$. If this set is empty, the next path from y to an input variable is considered, until $\Delta^\sharp$ changes or all paths have been explored. In the latter case, we say that y is *completely defined* in $\Delta^\sharp$; then, some other input variable must be chosen for refinement. Progress is guaranteed by eventually choosing a non-input variable that does not yet occur as a left-hand side in $\Delta^\sharp$. When there are no more such variables, $\Delta^\sharp$ has been completely refined into Δ.

The rationale of this *path refinement* strategy is to increase in a refinement step the number of paths in the graph G connecting a non-input variable to the input variables it ultimately depends on. This is more likely to produce a refinement that rules out spurious counterexamples than, say, a breadth-first refinement strategy over G, which may need to go through various levels of G before it starts adding constraints to $\Delta^\sharp$ that connect input and non-input variables.

The remaining question is which non-input variables to choose for refinement. For that, we rely on the help of the SMT solver. Let $\Delta^\sharp_{0,k}$ abbreviate $\Delta^\sharp_0 \wedge \cdots \wedge \Delta^\sharp_k$ and let $P_{0,k}$ abbreviate $P_0 \wedge \cdots \wedge P_k$. When the solver reports that

$$\Delta^\sharp_{0,k} \;\models_{\mathcal{IL}}\; \neg P_{0,k}$$

(the version of (1) using $\Delta^\sharp$) does not hold, it also returns a *counter-model* M for it. Specifically, M is a set of equalities of the form $x(m) = v$ where x is a stream variable, m is a numeral in $\{0, \ldots, k\}$, and v is a concrete value of the proper type (an integer number, a rational number, or a Boolean value), such that $M \models_{\mathcal{IL}} \Delta^\sharp_{0,k} \wedge \neg P_{0,k}$.

To check that M can be extended to a counterexample for the original program N, it is enough to check the $\mathcal{IL}$-satisfiability of the set $\Delta_0 \wedge \cdots \wedge \Delta_k \wedge M$.

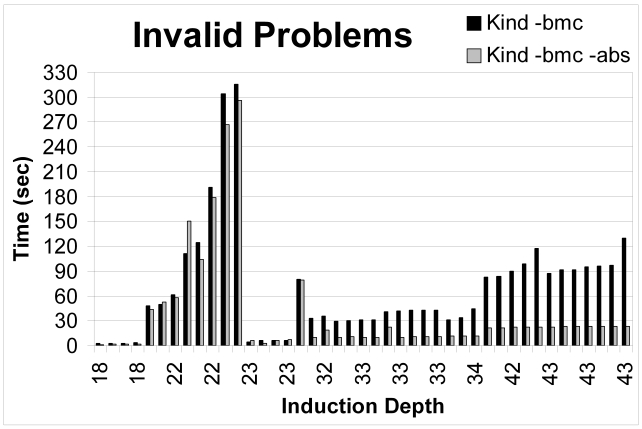

Figure 2: Runtimes for Kind with and without abstraction, on hard invalid problems.

If this set is $\mathcal{IL}$-unsatisfiable, $\Delta_n^\sharp$ needs to be refined. To direct the refinement to relevant program variables we ask the SMT solver for a minimal (or just smaller) subset M' of M that is enough to cause the unsatisfiability. Then, we pick for refinement a stream variable x that occurs in an equation $x(m) = v$ of M' and is not completely defined in $\Delta^\sharp$. This helps the refinement process focus on parts of the definition graph G whose absence from $\Delta^\sharp$ was decisive in generating the counter-model M.

4 Experimental Results

We implemented the k-induction framework above in *Kind*, a new automated inductive prover for idealized Lustre. Kind is written in OCaml and relies on an external SMT solver for reasoning in the logic $\mathcal{IL}$. Currently, Kind supports the SMT solvers CVC3 and Yices. The version discussed here uses Yices 1.0.9. Kind can be run in one of two major modes: *induction* and *bmc*. The first implements the k-induction decision procedure presented in Section 2.1. The second skips all operations related to the induction step of the procedure, making Kind essentially behave like a bounded model checker.

We performed extensive experiments with Kind, testing the performance of various combinations of enhancements over the basic induction procedure, and comparing the best ones against other verification systems.[4] The tests were run on a dedicated cluster of four 3.0 GHz Intel Pentium 4 machines, each with 1 Gb of physical memory, under RedHat Enterprise Linux 4.0. Timeouts were set to 900 seconds.

We based our test set on a number of sources. Most test problems were

[4]Kind's executable, the benchmarks discussed here as well as more detailed results are available from `http://combination.cs.uiowa.edu/Kind/`.

provided by colleagues in academia and industry. Others were adapted from published case studies. To increase the number of problems, we introduced a small number of modifications into the existing ones, simulating likely user errors. We grouped together all variants of the same original problem generated this way and removed any *duplicates* from the group, considering two variants duplicates if they produced too similar results (same valid/invalid answer and approximately the same runtimes) for all systems and configurations tried. This was done to prevent biasing the test set against or in favor of any one system.

The final test set had 1047 problems, each with a single property, classifiable into 6 groups: two groups with problems about memory controllers, one involving protocols, one based on simulations (often involving vehicles), one based on larger simulations, with several hundred lines of Lustre code apiece, and one with mostly toy problems involving counters. We partitioned the problems into: (*i*) a set of 447 *invalid problems*, problems whose property was disproved with a (real) counterexample; (*ii*) a set of 376 *valid problems*, problems whose property was declared proved by at least one systems; (*iii*) a set of 224 *unsolved problems*, which will not be considered below. Of the 823 solved problems, 20 valid and 18 invalid problems used rational streams, the rest used only integer and/or Boolean streams.

The most advanced system in the comparison was part of the SAL 3.0 toolset [9]. Even if it has an induction mode, strictly speaking, and contrary to Kind, SAL is not a full-blown induction prover. In induction mode, SAL first performs a BMC-like test up to a user specified limit l on the number k of unrollings, and then follows that with a *single* induction test if the BMC test found no counterexamples. For a fairer comparison, we replicated this behavior in Kind as well. Then we ran both SAL and Kind in an iterative deepening fashion on each problem, starting with $l = 0$ and repeatedly re-starting the system with a greater value of l until a conclusive answer was returned. We eventually chose to increment l by 3 at each restart because it seemed to produce the best results for both systems. For uniformity and space constraints, we report only results where iterative deepening was applied when Kind was run in induction mode.

Of the two main enhancements to the basic k-induction procedure presented in this paper, we focus on the evaluation of structural abstraction. The other enhancement, path compression, gave better overall results for Kind in induction mode, in terms of number of solved problems and run time. In particular, it allowed Kind to determine the validity of 8 problems that were not solvable by any of the other systems. Hence, all the results presented in this section are with Kind running with path compression on, unless specified otherwise. The other input options provided by Kind, and not discussed here, were each given the same value across the board.

4.1 Abstraction vs. no abstraction

When verifying safety properties, it is customary to attempt first a quick run of bounded model checking, to see if a counterexample can be found. If that

fails, the more expensive full verification check is then performed. Following this practice, it makes sense to look at Kind's results when run in bmc mode on the invalid problems and in interative-deepening inductive more on the valid ones.

Our expectation was that abstraction would not be effective with easy problems, because of its significant overhead. Hence, the purpose of our evaluation was to verify whether abstraction is beneficial with complex problems without producing an unacceptable slowdown with simple ones. Our results partially confirm this thesis.

As a simple complexity measure, let us classify a problem as *easy* if Kind could solve it within 2 seconds *without* abstraction, and *hard* otherwise.

Of the 447 invalid problems, 404 were easy in this sense and were cumulatively solved in 32s. Each of them had a counterexample path of length at most 13. The remaining problems took from 2s to 316s each to solve, for a total of 2989s, with counterexample lengths ranging from 18 to 43. With abstraction, Kind solved each of the easy problems within 2s as well, but took 73s overall, with a slowdown factor of 2.3. However, it solved the hard problems in 1694s with an overall speed up factor of 1.8. Runtimes in seconds for the hard invalid problems are plotted in Figure 2. As can be clearly seen, only in one case does abstraction produce a significant slowdown. In most of the other cases it is instead quite effective, with speed-ups in excess of 300% for several problems.

In contrast, abstraction was not effective for the valid problems. To start, almost all of them, 370 over 376, were easy and were solved in a total of 35s without abstraction, with induction depths (the value of k) of 12 or less. The 6 hard problems took from 2s to 101s, with one timeout. The total time for them was 217s, with induction depths ranging from 11 to 17. With abstraction, Kind solved each of the easy problems within 2s except six (solved in less than 4s), with a total time of 115s and a slowdown factor of 3.3. It also solved all of the hard problems, but it was actually 1.6 times slower on them (340s) than Kind without abstraction.

4.2 Kind vs. other Lustre checkers

To evaluate Kind against the state of the art in the automated verification of Lustre programs, we compared it with all the publicly-available Lustre verifiers we were aware of: Lesar, built by Lustre's developers at Verimag [20], Nbac, also developed at Verimag [18], and Rantanplan and Luke, both developed at Chalmers University [13, 8]. We were unable to run the latest version of Nbac, and we used instead the version provided in the Lustre 4 distribution. Unfortunately, this version appears to be unsound[5] and so we had to discard it from this evaluation.

Lesar is a state-based verification tool included with the Lustre 4 distribution. It is primarily a BDD-based symbolic model checker, with some limited

[5]It classified as valid a number of problems for which at least two of the other systems were able to find a counterexample.

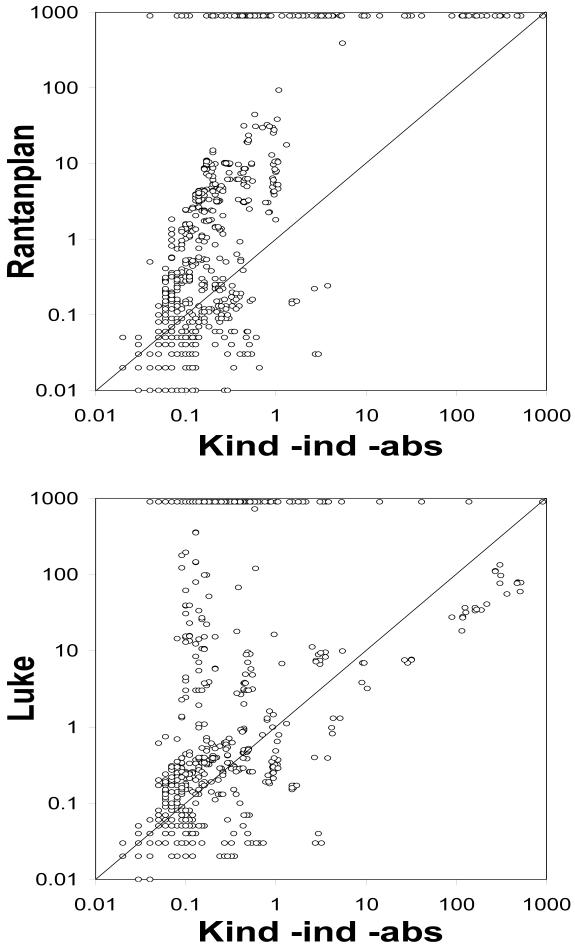

Figure 3: Kind vs. Rantanplan and Luke on valid and invalid problems.

support for integers through abstraction and a polyhedral library. Due to this, it is incomplete for non-Boolean problems.

Rantanplan could be considered the closest precursors to Kind since it is based on k-induction and SMT techniques. As mentioned in related work, its induction procedure and its SMT support are however less sophisticated, in particular it does not perform abstraction and refinement. Finally, it does not support programs with rational streams.

Luke is another k-induction verifier, inspiring much of the work in Rantanplan, but it is based on propositional logic and was developed mostly for educational purposes. It accepts programs with integers by treating them as bounded integers of a user-specified size.

Since none of these systems have something comparable to Kind's bmc mode,

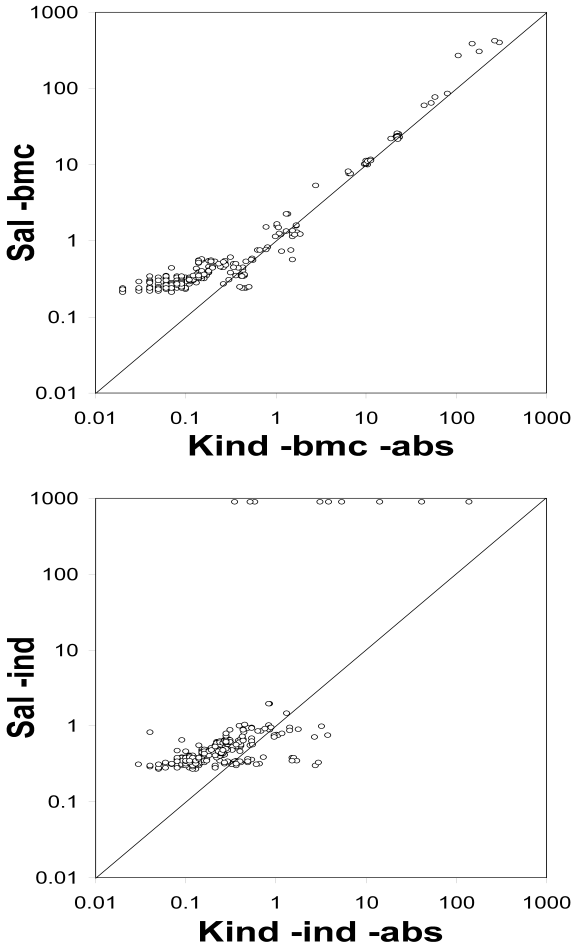

Figure 4: Kind vs. SAL, respectively on invalid and valid problems.

we report the results obtained by Kind when run in iterative-deepening induction mode on both valid and invalid problems. In that mode, Kind solved all of the valid problems but timed out on 8 of the invalid ones.

We ran Lesar from the distribution in its standard configuration (with the -poly argument). Lesar solved correctly fewer than 10% of the valid problems, often quickly producing an incorrect answer for the rest. Since Lesar is incomplete but does not return counterexamples, measuring its performance on the invalid problems is not meaningful.

Rantanplan and Luke were more comparable to Kind. Their performance against our system is summarized in the scatter plots of Figure 3, which show runtimes in seconds, on a log-log scale. Timeout points also include problems that a system solved incorrectly due to its incompleteness, or could not solve

because they were out of its scope or caused a runtime error (such as stack overflow).

We tried several of the configurations for Rantanplan suggested in [12], with generally similar results, and chose one that seemed one of the best overall performers: deletion filtering with the Pooh checker used in online integration and GLPK as the offline infeasibility checker. Rantanplan solved 84% of the 823 solved problems, (against Kind's 99%) and was overall an order of magnitude slower than Kind on them. Of the 133 problems not solved by Rantanplan, 38 could not be solved due to presence of rational streams, 2 produced incorrect counterexamples, 26 caused run-time errors, and the rest timed out.

Consistently to what observed in [12], Luke performed better than Rantanplan on the invalid problems. However, it solved only 71% of the 823 solved problems (mostly finite-state problems), being overall 30% faster than Kind on those. Of the 242 problems not solved by Luke, 38 could not be solved due to presence of rational streams, 19 produced incorrect counterexamples due to the incompleteness of the bounded integer support, and the rest timed out.

4.3 Kind vs. SAL

Even if Kind was consistently and significantly better than the available Lustre-native verifiers, it could be argued that those systems are, for various reasons, not cutting-edge anymore. We looked then for a publicly available state-of-the-art tool that could verify safety properties of both finite- and infinite-state reactive systems. The eventual choice of the SAL toolset was motivated as follows.

The toolset contains a model checker (sal-inf-bmc, hereafter referred to as SAL) in many ways similar to Kind: it can be run in either bmc or induction mode, has a form of path compression, is SMT-based, and uses Yices. The main differences are that it does not use abstraction and is not Lustre-specific. Instead, it uses its own input language, based on the traditional two-state model. This allowed us to verify our hypothesis that adapting existing techniques to work directly on a logical model of Lustre leads to better performance.

To run SAL on Lustre problems we utilized a well-tuned Lustre-to-SAL translation developed at Rockwell Collins along the lines of the translations described in [22]. Rockwell Collins has been developing and continually improving translators from Lustre to various model checkers, including SAL, over several years for the verification of production-level Lustre models of its avionics software.

We ran SAL and Kind in their respective bmc mode on the invalid problems, and in iterative-deepening induction mode on the valid ones. Among SAL's different command-line configurations only its analogous to path compression (the -acyclic option) produced any significant differences in overall performance with respect to the default configuration. Specifically, enabling path compression was better in induction mode and worse in bmc mode for SAL. The scatter plots of Figure 4 summarize SAL's results in those configurations against Kind's results with path compression enabled only in induction mode and abstraction enabled

in both modes. As the plots show, the two systems are comparable but Kind's performance dominates.

Both systems solved all the invalid problems, with Kind being more than 50% faster overall. Moreover, Kind solved all valid problems while SAL timed out on 10 of them. On the valid problems solved by both systems, Kind was more than 50% faster. Interestingly, on the latter problems, Kind without abstraction, a configuration more closely comparable to SAL, was actually more than 500% faster.

Overall, these results seems to support our hypothesis that a Lustre-specific k-induction tool would offer a performance premium over a translation-based approach. On the other hand it must be added that, looking at its source code, SAL does not appear to exploit the advanced features of Yices as much as Kind does, which might be a big factor in Kind's better performance.

5 Conclusion and Further Work

We have presented a general approach for the verification of Lustre programs built around SMT-based k-induction. The approach has a wider scope than that of existing verifiers for Lustre since it can handle finite- and infinite-state programs with integer and rational streams. It scales much better to realistic programs thanks to the compactness of its logical translation and a tight integration with efficient SMT solvers. Also, it lends itself quite naturally to a form of structural abstraction and refinement that adds further scalability when model checking complex properties or programs. The approach seems general enough to be applicable to other synchronous dataflow languages as well.

We conclude by mentioning a couple of additional directions for further work. Our path refinement strategy adds to the current abstraction entire equations from the concrete system Δ. More fine-grained refinements would be obtained by statically preprocessing Δ and breaking complex equations it into simpler ones by introducing additional local streams. We would like to investigate and evaluate automated strategies for doing this.

Our abstraction technique can support modular verification of multinode programs when individual nodes are annotated with sufficiently expressive invariants. We plan to investigate ways of avoiding or delaying the in-lining of subnode calls when invariants are available. We also plan to investigate ways to produce node invariants automatically during the verification of a property, with the goal of speeding up later re-verification attempts of the same property caused by changes in the code or the property's formulation—something that occurs often in real world situations.

Acknowledgments. This work was partially supported by an equipment grant from Intel Corporation. We thank Koen Claessen for some discussions on k-induction and its implementation in Luke; Bruno Duterte for his assistance with Yices; Anders Franzén for providing many of the problems in our test set. Special thanks go to Steve Miller and Mike Whalen for providing additional problems and translating all of them to SAL. We are also grateful to

the anonymous reviewers for some useful suggestions on improving the paper's presentation.

References

[1] J. M. A. Armando and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Proceedings of SPIN'06*, volume 3925 of *LNCS*, pages 146–162. Springer, 2006.

[2] D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *Proceedings of CAV 2007*. pages 366–378. Springer, 2007.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[4] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Handbook on Satisfiability*, chapter Satisfiability Modulo Theories. IOS Press, 2008. (To appear.).

[5] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Proceedings of FMCAD 2000*, pages 372–389, Springer, 2000.

[6] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In M. Young, editor, In *Proceedings of ISSTA 98*, pages 102–112. ACM, 1998

[7] P. Chauhan *et al.* Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Proceedings of FMCAD'02* , pages 33–51. 2002.

[8] K. Claessen. Luke webpage, 2006. `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/form/luke.html`.

[9] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Proceedings of CAV 2004*. Springer, 2004.

[10] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *Proceedings of CAV 2003*, volume 2725 of *LNCS*, 2003.

[11] N. Een and N. Sorensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.

[12] A. Franzén. Combining SAT solving and integer programming for inductive verification of Lustre programs. Master's thesis, Chalmers University of Technology, 2004.

[13] A. Franzén. Using satisfiability modulo theories for inductive verification of Lustre programs. *Electronic Notes in Theoretical Computer Science*, 114, 2005.

[14] M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *Proceedings of ICCAD'06*, pages 794–801. ACM, 2006.

[15] G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, The University of Iowa, 2008. (Forthcoming).

[16] 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.

[17] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by mans of the synchronous data-flow language LUSTRE. *IEEE Transactions in Software Engineering*, 18(9):785–793, 1992.

[18] B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of synchronous programs. *Formal Methods in System Design*, 23:5–37, 2003.

[19] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[20] G. Pace, N. Halbwachs, and P. Raymond. Counterexample generation in symbolic abstract model-checking. *International Journal on Software Tools and Technology Transfer (STTT)*, 5(2):158–164, 2004.

[21] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of FMCAD 2000*, pages 108–125. Springer, 2000.

[22] M. Whalen, D. Cofer, S. Miller, B. Krogh, and W. Storm. Integration of formal analysis into a model-based software development process. In *12th International Workshop on Industrial Critical Systems (FMICS 2007)*, 2007.