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

George Hagen and Cesare Tinelli
The University of Iowa

Outline

- Background
- Research Contributions
- Experimental Results
- Conclusion

Deductive Verification vs. MC

Deductive Verification

- Pros
  - Natural translation
  - Unrestricted data types
  - Arbitrary properties
  - Favors proving validity
- Cons
  - Time consuming
  - Expertise required
  - May be hard to produce counterexamples

Model checking

- Pros
  - Fast
  - Automatable
  - Generates concrete counterexamples
- Cons
  - More complex translation
  - Finite data types only
  - Propositional properties
  - Harder to prove validity

Main Idea of This Work

- Middle-ground approach
- Use SMT-based model checking:
  - Automatically translate transition relation T and property P into a first-order logic (FOL) specification language
    - Decidable fragment of FOL supported by SMT solvers
      - Uninterpreted functions
      - Linear arithmetic
      - Arrays, Tuples, Records
  - Try to prove or disprove P automatically with an inductive model checker/verifier

Satisfiability Modulo Theories (SMT) [64, 62]

- Lifting of Boolean techniques to include decidable fragments of data type theories
- Use efficient reasoners to handle non-Boolean terms
- SAT $\rightarrow$ SMT
  - Boolean formulas $\rightarrow$ quantifier free first order formulas
  - More powerful than Boolean representation, but retain decidability
  - More compact formulas, better scalability
  - More natural translations

k-induction to Verify Safety Properties

- SMT + induction to verify property P
- Strengthen by increasing timeframe examined:

base:

$$I(S_0) \wedge T(S_0, S_1) \wedge \ldots \wedge T(S_{k-1}, S_k) \models P(S_0) \wedge \ldots \wedge P(S_k)$$

step:

$$T(S_n, S_{n+1}) \wedge \ldots \wedge T(S_{n+k}, S_{n+k+1}) \wedge P(S_n) \wedge \ldots \wedge P(S_{n+k}) \models P(S_{n+k+1})$$

  - If step does not hold: increase k
- Note:
  - Base formula & step formulas are SMT formulas
  - Base case is just BMC

So…

- We can use SMT-based k-induction to verify safety properties of transition systems

- We are interested in reactive systems, often described with synchronous dataflow languages, such as Lustre

Lustre Example

```
node thermostat (a_temp, t_temp, marg: real) returns (cool, heat: bool) ;
  let
      cool = (a_temp - t_temp) > marg ;
      heat = (a_temp - t_temp) < -marg ;
  tel

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

Lustre Language

- Structure
  - Stream definitions - equations
  - Nodes - programs as stream definition macros
- Basic types (of streams):
  - Boolean, integer, real
- Complex types:
  - Tuples, supplemental array, record data structures
- Operators
  - (mostly) lifting of Boolean & arithmetic operators to streams
  - Temporal operators: **pre**, **->**, **when**, **current**

Lustre [17,42,43]

- Lustre is an equational synchronous dataflow language
- System of equational constraints between input and output streams
- We can model a stream s of values of type τ as a function

  $$s : \mathbb{N} \rightarrow \tau,$$

  that maps instants to stream values
- Functional, in the sense of no side effects

Lustre

- Stream constraints can be reduced to Boolean & arithmetic constraints over instantaneous configurations:

$$\begin{cases} margin(n) = 1.5 \\ target(n) = ite(n = 0, 70.0, ite(dn(n), target(n-1) - 1.0, \ldots)) \\ cool(n) = (actual(n) - target(n)) > margin(n) \\ heat(n) = (actual(n) - target(n)) < (-margin(n)) \end{cases}$$

- **Crucial observation**: SMT solvers can process these sorts of constraints

Outline

- Background
- Research Contributions
- Experimental Results
- Conclusion

Research Contributions

- Translation of Lustre program + properties into SMT representation Idealized Lustre logic ($\mathcal{IL}$)

- Use SMT-based *K*-induction to prove invariant properties of Lustre programs
  - Enhance with path compression, abstraction, other optimizations.

Idealized Lustre Logic ($\mathcal{IL}$)

- First order language with built-in
  - Linear integer arithmetic
  - Linear real arithmetic
  - Tuples

Lustre program as $\mathcal{IL}$ constraints

- Lustre code

```
node alarm_timer (reset: bool; x,a: int) returns (signal: bool);
var time, alarm: int;
let
    time = x -> if reset then x else pre(time)+1;
    alarm = a -> if reset then a else pre(alarm);
    signal = (time = alarm);
tel
```

- $\mathcal{IL}$ constraints

$$\Delta_n = \begin{cases} time(n) = ite(n = 0, x(0), ite(reset(n), x(n), time(n-1)+1)) \\ alarm(n) = ite(n = 0, a(0), ite(reset(n), a(n), alarm(n-1))) \\ signal(n) = time(n) = alarm(n) \end{cases}$$

- Property: $P_n = (\neg signal(n) \Rightarrow time(n) < alarm(n))$

From Programs to Idealized Lustre Logic $\mathcal{IL}$

- N be a single-node Lustre program
- N's stream variables: $\mathbf{v} = < x_1, \ldots, x_m, y_1, \ldots, y_q >$

$$\Delta_n = \begin{cases} y_1(n) = t_1[\mathbf{v}(n), \mathbf{v}(n-1), \ldots, \mathbf{v}(n-d)] \\ \vdots \\ y_q(n) = t_q[\mathbf{v}(n), \mathbf{v}(n-1), \ldots, \mathbf{v}(n-d)] \end{cases}$$

- d is memory depth of N
- Nodes can be seen as macros & inlined

SMT-based k-induction in $\mathcal{IL}$

- To check P is invariant, find k such that:

$$\text{base}:$$

$$\Delta_0 \wedge \ldots \wedge \Delta_k \models_{\mathcal{IL}} P_0 \wedge \ldots \wedge P_k$$

$$\text{step}:$$

$$\Delta_n \wedge \ldots \wedge \Delta_{n+k+1} \wedge P_n \wedge \ldots \wedge P_{n+k} \models_{\mathcal{IL}} P_{n+k+1}$$

- $\models_{\mathcal{IL}}$ decided by an SMT solver for $\mathcal{IL}$

k-induction may not be enough

Reasons:

i. P might be a non-inductive invariant property

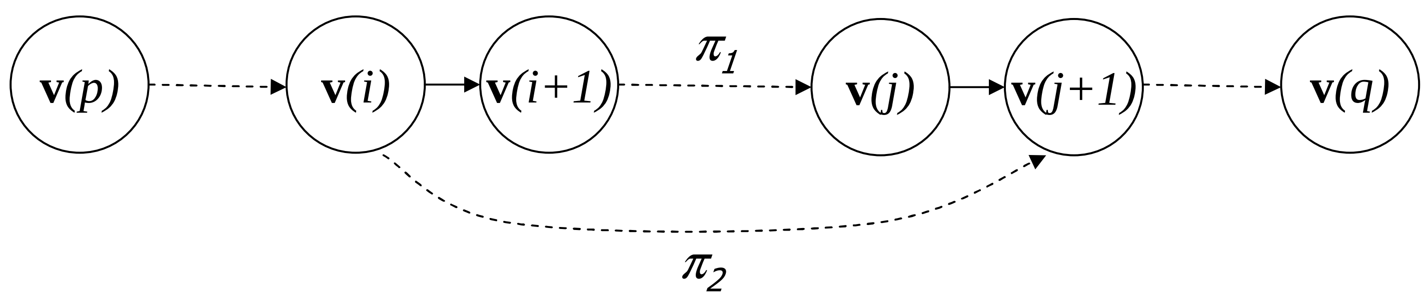
ii. Basic k-induction may be too expensive

Enhancements to *K*-induction algorithm

1. Path compression (i)
2. Termination check (i)
3. Abstraction (ii)

1. Path Compression (i)[32]

- Invariant strengthening technique
- Enforces distinct configurations
  - Reduced set of "memory"/state variables
- If state variables $\mathbf{x}_i = \mathbf{x}_j$ for configurations i and j, then we may compress configurations $i+1$ through j



2. Termination check (i) [63]

- Same idea as path compression
- If all concrete paths of length $k+1$ have cycles, then we have explored the reachable space, and may terminate
- Can prove some non-inductive properties

k-induction with Path Compression

base:

$$\Delta_0 \wedge \ldots \wedge \Delta_k \models_{\mathcal{IL}} P_0 \wedge \ldots \wedge P_k$$

step:

$$\Delta_n \wedge \ldots \wedge \Delta_{n+k+1} \wedge P_n \wedge \ldots \wedge P_{n+k} \wedge C_{n,k} \models_{\mathcal{IL}} P_{n+k+1}$$

termination check:

$$\Delta_0 \wedge \ldots \wedge \Delta_k \models_{\mathcal{IL}} \neg C_{0,k+1}$$

k-induction with Path Compression

base:

$$\Delta_0 \wedge \ldots \wedge \Delta_k \models_{\mathcal{IL}} P_0 \wedge \ldots \wedge P_k$$

step:

$$\Delta_n \wedge \ldots \wedge \Delta_{n+k+1} \wedge P_n \wedge \ldots \wedge P_{n+k} \wedge C_{n,k} \models_{\mathcal{IL}} P_{n+k+1}$$

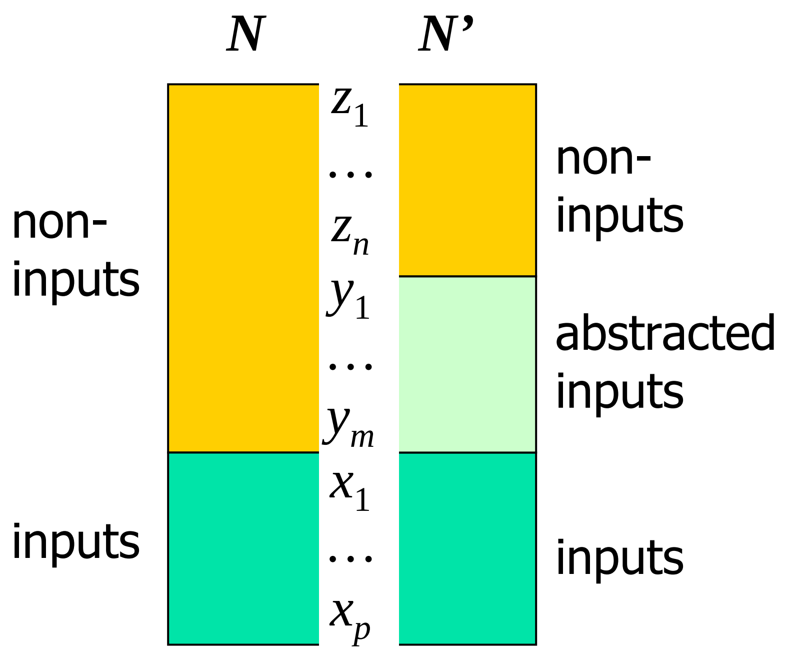
termination check:

$$\Delta_0 \wedge \ldots \wedge \Delta_k \models_{\mathcal{IL}} \neg C_{0,k+1}$$

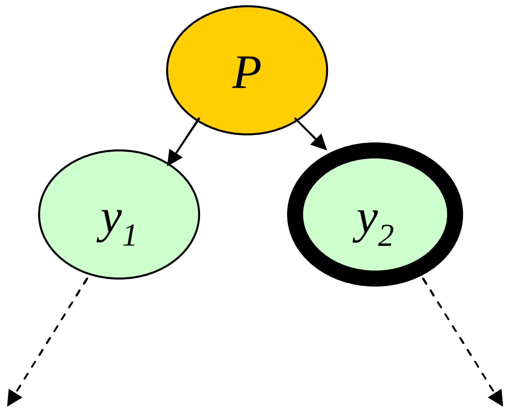
Compression constraint

3. Abstraction/Refinement in Lustre (*ii*)

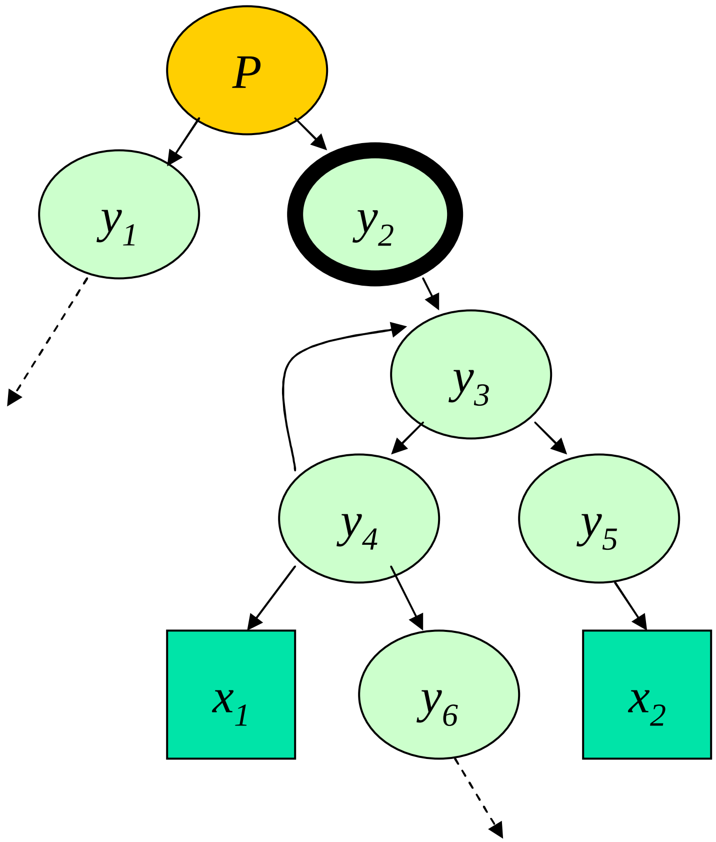
- Let N be a Lustre program
- Over-approximate N with N' by treating some of N's non-input streams as input
- Initial abstraction only contains definitions of stream variables in property ($\mathbf{z}$)
- Refine abstraction by adding definitions of variables in $\mathbf{y}$
- CEGAR / structural abstraction [24,52,18,4]



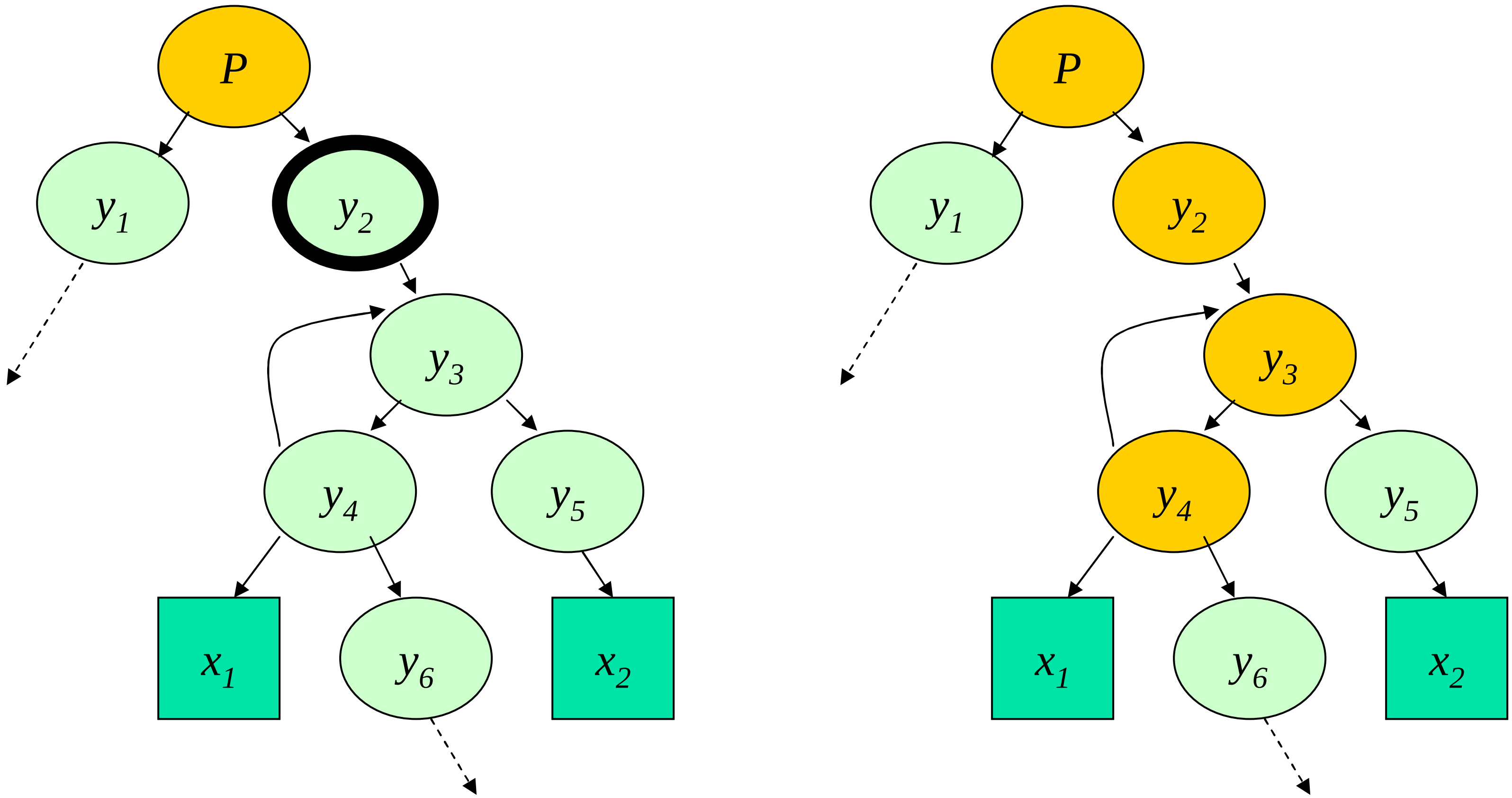
"Path" Refinement Example



"Path" Refinement Example



“Path” Refinement Example



k-induction with Abstraction

$$\text{base}:$$

$$\Delta'_0 \wedge \ldots \wedge \Delta'_k \models_{\mathcal{IL}} P_0 \wedge \ldots \wedge P_k$$

$$\text{step}:$$

$$\Delta'_n \wedge \ldots \wedge \Delta'_{n+k+1} \wedge P_n \wedge \ldots \wedge P_{n+k} \models_{\mathcal{IL}} P_{n+k+1}$$

- Also checking for & eliminating spurious counterexamples
  - Done in base & inductive cases

Outline

- Background
- Research Contributions
- Experimental Results
- Conclusion

KIND solver

- We built a new verifier implementing these ideas
- Uses Yices / CVC3 SMT solvers
- May be run in BMC mode or induction mode
- Comparisons with existing tools: Lesar, Luke, Rantanplan, SAL

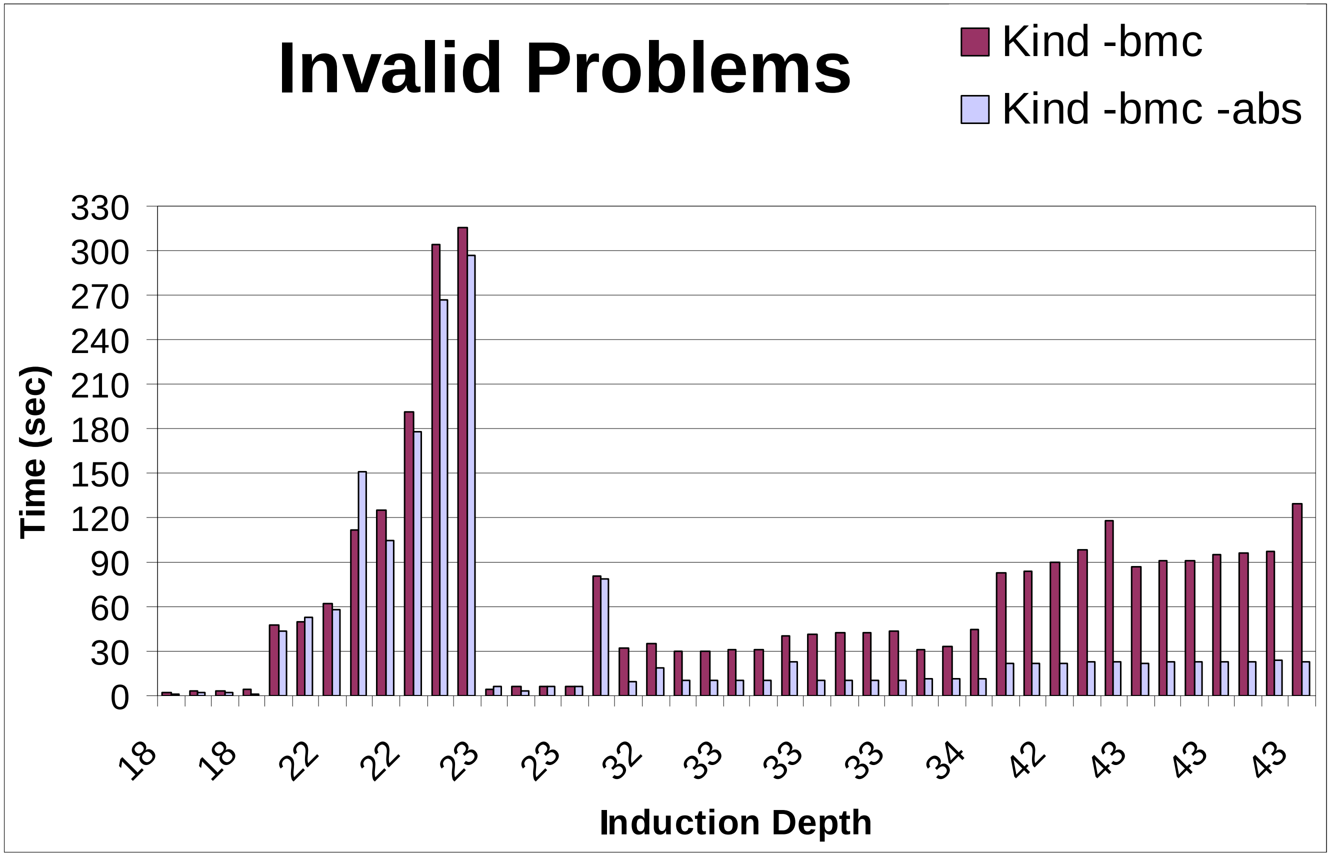
Problem set

- 1047 problems
  - Hand-crafted Lustre examples
  - Published industrial case studies
  - Rockwell Collins examples
  - 376 Valid, 447 Invalid, 224 Unsolved
- Timeouts
  - >900 sec
  - Program abort
  - Incorrect counterexample (incomplete)

Results: Impact of Enhancements

- Abstraction
  - invalid (BMC) cases: ~2x speedup overall
  - valid cases: ~2x slowdown (extra overhead)
- Path compression + Term. check:
  - Solved 29/376 more problems in valid cases (including all “hard” problems)
- Termination check:
  - Kind solved 8/376 more problems than other systems
  - High overhead for BMC/invalid problems (~10x slowdown)

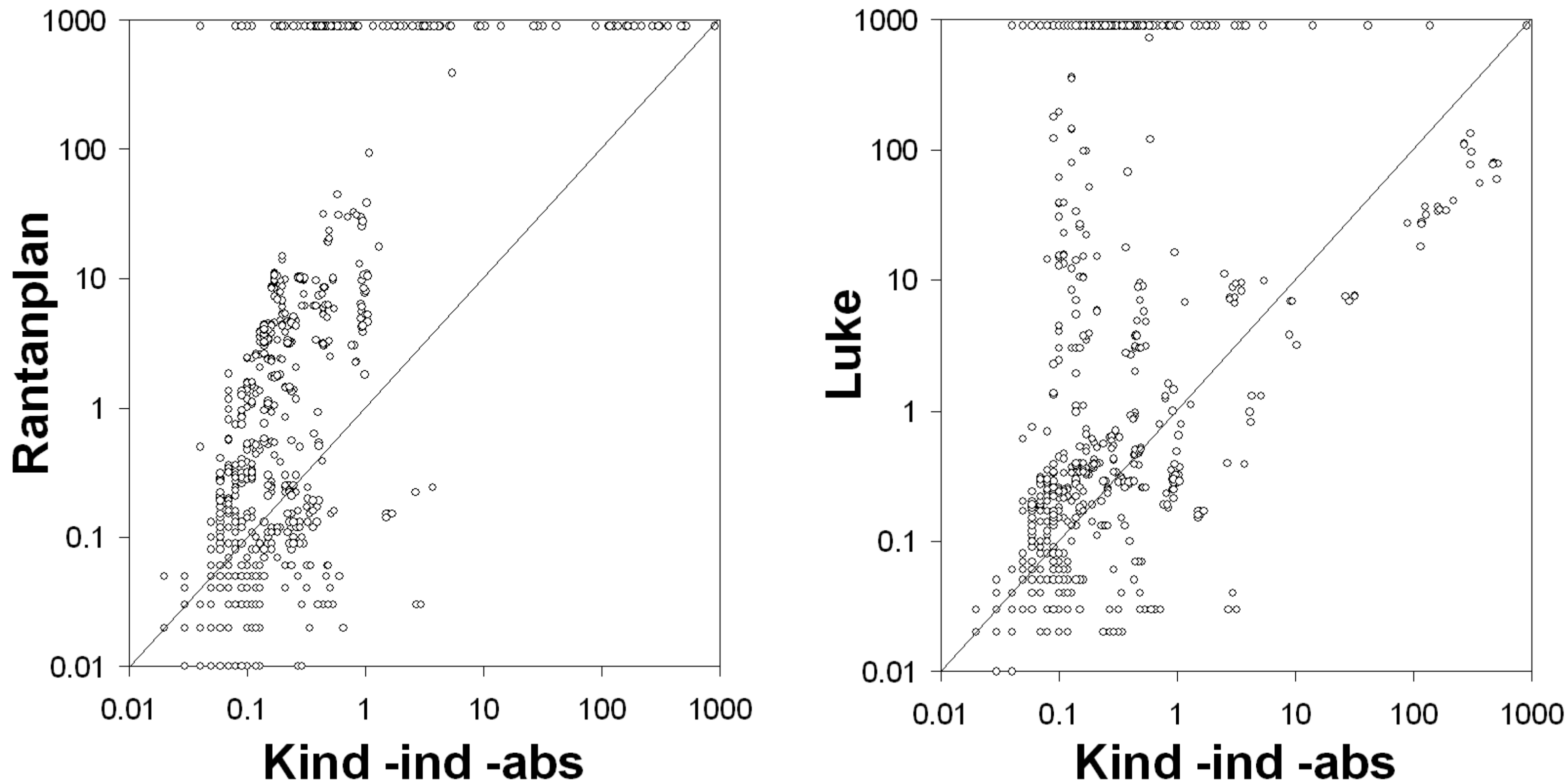
Abstraction vs. Non-abstraction (hard invalid problems)



Comparison w/ Other Systems

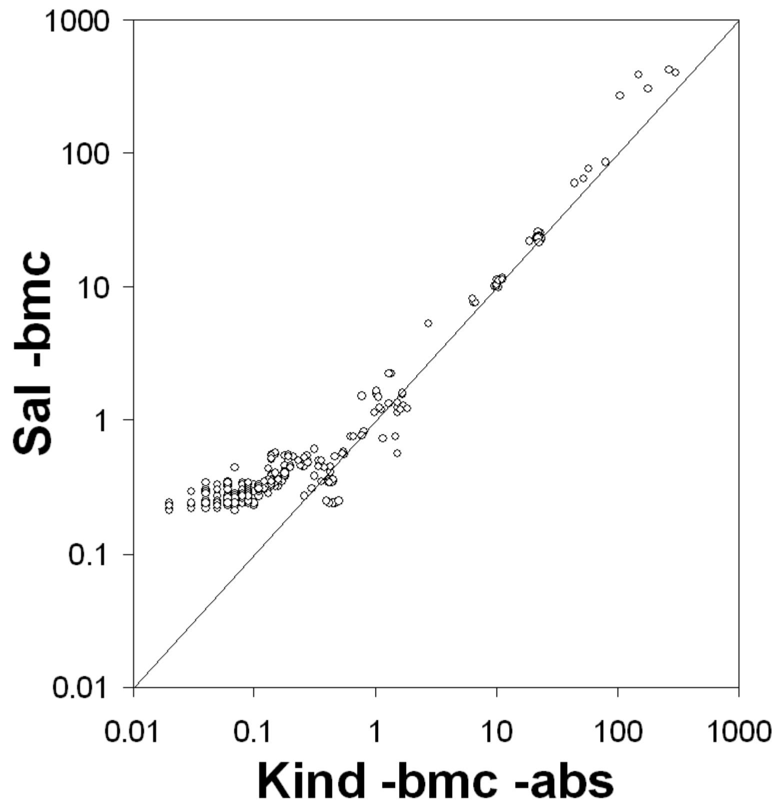
- Luke [22], Rantanplan [38, 39] (Chalmers)
  - Inductive model checkers
  - Rantanplan adds SMT (supports ILP only)
- SAL (SRI) [31, 65]
  - sal-inf-bmc inductive 2-state model checker
  - Rockwell Collins translations to SAL [73]
- …

Comparative Results (Rantanplan & Luke)

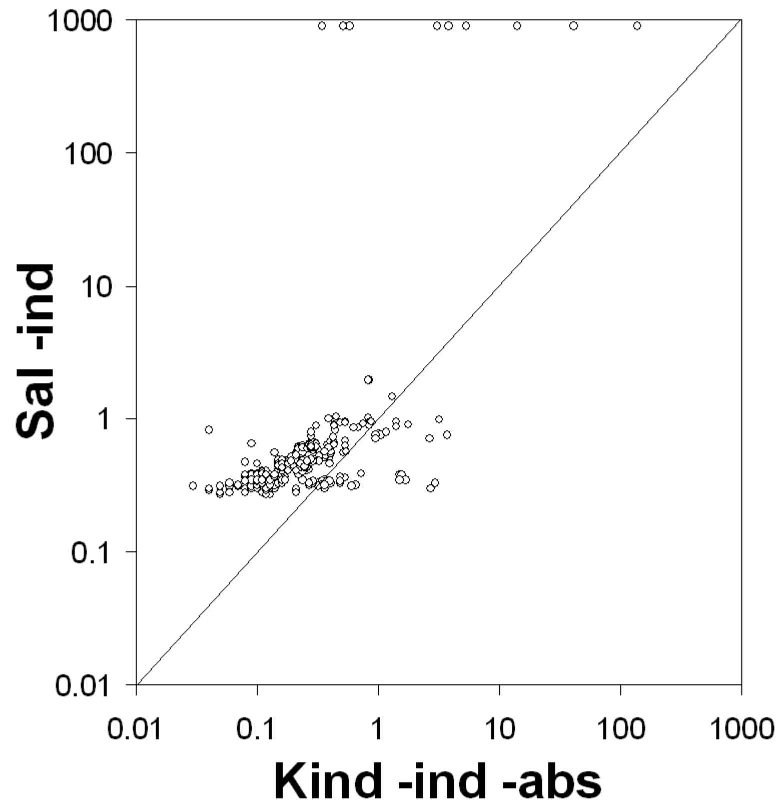


All solved problems

Comparative Results (SAL)



Invalid problems



Valid problems

Conclusion

- Translation of Lustre program + properties into suitable first order logic $\mathcal{IL}$ with built-in theories
- Used off-the-shelf SMT solvers to prove safety properties of Lustre programs with k-induction
  - Enhanced with path compression & abstraction
- Highly competitive with state of the art systems

Future Work

- Structural abstraction variants
- Modular verification
- Support for nonlinear algebra

Works Referenced

[4] Domagoj Babic and Alan J. Hu. Structural abstraction of software verification conditions. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, pages 366–378. Springer, 2007.

[17] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACTSIGPLAN symposium on Principles of programming languages (POPL '87)*, pages 178–188, New York, NY, USA, 1987. ACM.

[18] William Chan, Richard J. Anderson, Paul Beame, and David Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In Michal Young, editor, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 98)*, pages 102–112, Clearwater Beach, Florida, USA, March 1998. ACM.

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

[24] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, 2000.

Works Referenced

[31] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. Tool presentation: SAL 2. In *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV 2004)*. Springer-Verlag, 2004.

[32] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, 2003.

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

[39] Anders Franzén. Using satisfiability modulo theories for inductive verification of Lustre programs. In *Third International Workshop on Bounded Model Checking (BMC 2005)*, volume 114 of *Electronic Notes in Theoretical Computer Science*, pages 19–33. Elsevier, Jan 2005.

[42] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[43] Nicholas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions in Software Engineering*, 18(9):785–793, 1992.

Works Referenced

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

[62] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 3:141–224, 2007.

[63] Mary Sheeran, Satnam Singh, and Gunnar St°almarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD '00)*, pages 108–125, London, UK, 2000. Springer-Verlag.

[64] Hossein M. Sheini and Karem A. Sakallah. From propositional satisfiability to satisfiability modulo theories. In *9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, pages 1–9, 2006.

[65] SRI International. Symbolic Analysis Laboratory webpage. http://sal.csl.sri.com.

[73] Michael Whalen, Darren Cofer, Steven Miller, Bruce Krogh, and Walter Storm. Integration of formal analysis into a model-based software development process. In *12th International Workshop on Industrial Critical Systems (FMICS 2007)*, Berlin, Germany, July 2007.

[74] Mike Whalen. Autocoding tools interim report. Rockwell Collins internal report, 2004.

Some Terminology

	var \ instant	0	1	2	3	4	…
Inputs	x_1	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	…
	$\vdots$			$\vdots$			
	x_l	$a_{l,0}$	$a_{l,1}$	$a_{l,2}$	$a_{l,3}$	$a_{l,4}$	…
Non-inputs	y_1	$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$	$b_{1,4}$	…
	$\vdots$			$\vdots$			
	y_m	$b_{m,0}$	$b_{m,1}$	$b_{m,2}$	$b_{m,3}$	$b_{m,4}$	…
	p	t	t	t	t	f	…

Key:

- Instantaneous configuration
- Trace
- Path
- Counterexample