Scalable and Accurate Verification of Data Flow Systems

Cesare Tinelli
The University of Iowa

Overview

- AFOSR Supported Research
- Collaborations
  - NYU (project partner)
  - Chalmers University (research collaborator)
  - Rockwell-Collins (end user)
- Overall Objective
  - Improve performance and scalability of techniques for verifying mission-critical embedded software

Overview

- Scientific Approach
  - Focus on synchronous data-flow systems
  - Develop automated reasoning techniques and tools based on more powerful logic than propositional logic
  - Develop modular and compositional verification methods and tools
- Breakthrough Opportunities
  - Use Satisfiability Modulo Theories (SMT) instead of SAT
  - Exploit recent dramatic advances in SMT technology
  - Reason about finite- as well as infinite-state systems

Formally Verifying Software Systems

Traditional main alternatives:

- Deductive verification

  Based on first- or higher-order logical calculi for theorem proving

- Model checking

  Based on automata or SAT techniques

Deductive Verification vs Model Checking

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
- SMT-based model checking:
  - Automatically translate system S and property P into a first-order logic with built-in theories
  - Try to prove or disprove P automatically with an inductive model checker/verifier
  - Use an SMT solver as the main reasoning engine
  - Verify control and data properties

Satisfiability Modulo Theories

- Lifting of SAT solving techniques to certain fragments of data type theories (linear arithmetic, arrays, lists, tuples, …)
- Uses efficient specialized reasoners to handle predicates with theory symbols (>, +, a[i], cons, …)
- SAT → SMT
  - Boolean formulas → quantifier free formulas over theories
  - More powerful than Boolean representation, but retains decidability and efficiency
  - More compact formulas, better scalability
  - More natural encodings for verification

This Research

Background

- Seminal work on SAT-based temporal induction at Chalmers University

  [Biesse&Claessen, Sheeran *et al.*]

Our contributions

- lifting to SMT case
- extension to infinite-state systems
- enhancements to induction method
- state-of-the-art SMT solver

Main Focus and Approach

- Consider reactive systems specifiable with synchronous dataflow languages
- Use SMT-based k-induction to verify safety properties (ie., invariant functional properties) of transition systems
- For experimental evaluations, work with systems written in Lustre

Essence of Lustre Programs

- Declarative and deterministic
- System of equational constraints between input and output streams
- Each stream s of values of type τ is a function

$$s : ☠ \rightarrow \tau$$

that maps instants to stream values

Lustre by Example

```
node thermostat (a_temp, d_temp, marg: real) returns (cool, heat: bool)
   let
         cool = (a_temp - d_temp) > marg ;
         heat = (a_temp - d_temp) < -marg ;
   tel
node therm_control (actual: real; up, dn: bool) returns (heat, cool: bool)
var target, margin: real;
   let
         margin = 1.5 ;
         desired = 21.0 -> if dn then (pre desired) - 1.0
                                 else if up then (pre desired) + 1.0
                                 else pre target ;
         (cool, heat) = thermostat (actual, desired, margin) ;
   tel
```

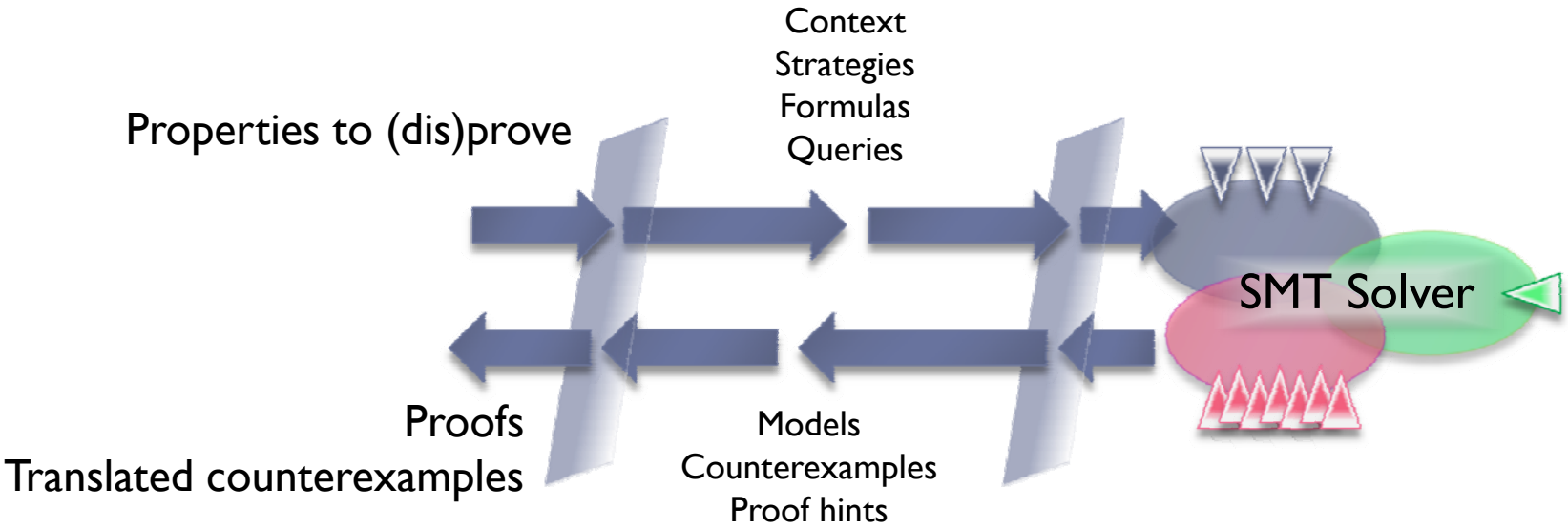
From stream algebra constraints to temporal constraints

- 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 this sort of constraints natively and efficiently

SMT Solvers

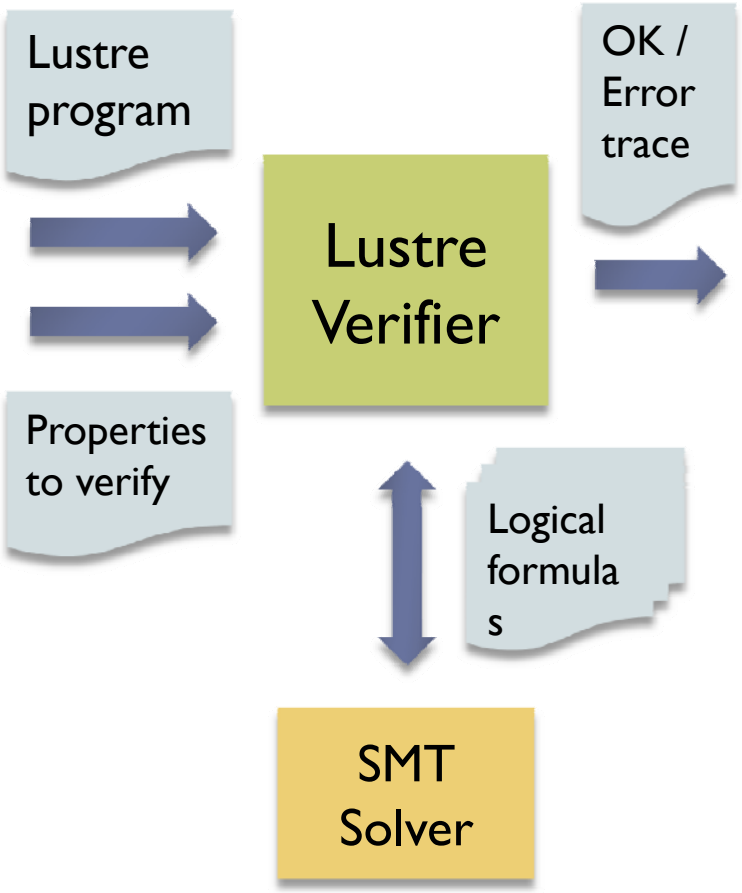


- SMT Solver: automatic engine for checking satisfiability/validity
- Formulas can encode transition systems and their properties
- Solver returns with:
  - Whether input formula is valid or not in the built-in theory
  - If so, a proof of validity
  - If not, a counter-model

Our Verification Framework

- Translation of
  - a Lustre program L and
  - a putative invariant property P

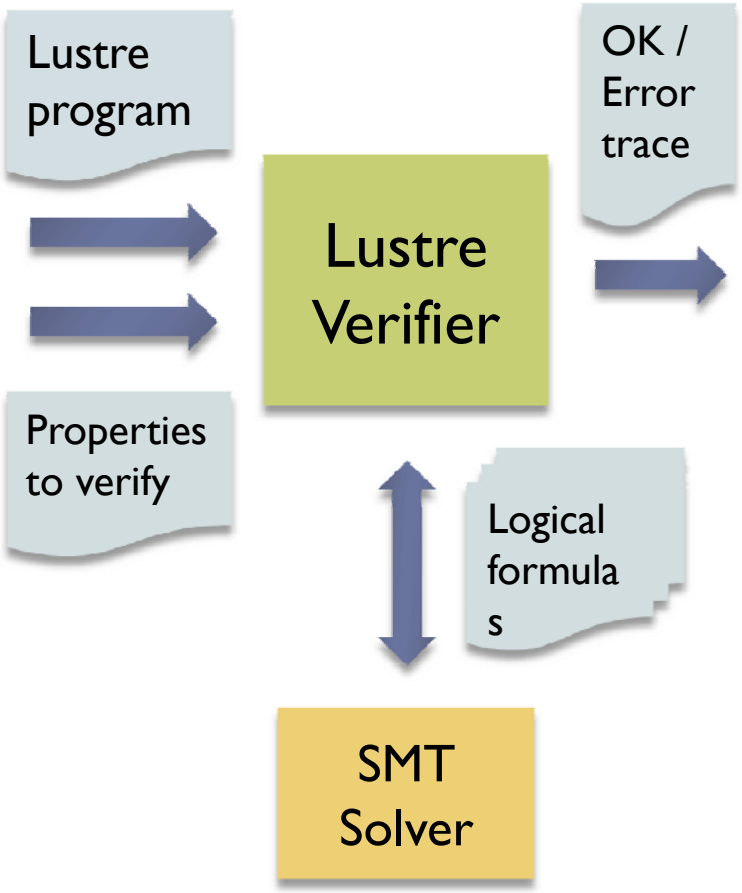
  into set F of SMT formulas
- SMT-based k-induction on F to prove or disprove P for L
- Main enhancements:
  - path compression
  - abstraction
  - invariant discovery



Our Verification Framework

- Translation of
  - a Lustre program L and
  - a putative invariant property P

  into set F of SMT formulas
- SMT-based k-induction on F to prove or disprove P for L
- Main enhancements:
  - path compression
  - abstraction
  - invariant discovery



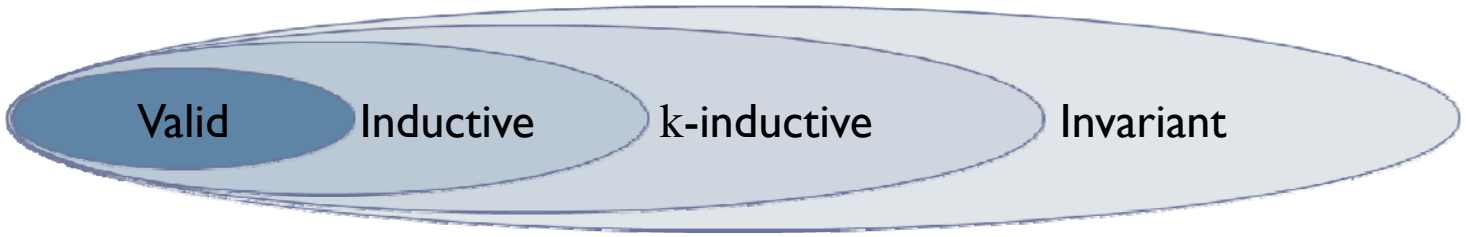
Some Basic Terminology

- Q : a state space
- S : a state transition system over Q
- I : set of S's initial states
- T : S's transition relation

- Reachable states: all initial states of S and
  all T-successors of reachable states

A Classification of Functional Properties

- Valid:
  - satisfied by all states in Q
- Inductive:
  - $I(s_0) \Rightarrow P(s_0)$
  - $P(s_n), T(s_n, s_{n+1}) \Rightarrow P(s_{n+1})$
- k-inductive:
  - $I(s_0), T(s_0, s_1), \ldots, T(s_{k-1}, s_k) \Rightarrow P(s_0), \ldots, P(s_k)$
  - $T(s_n, s_{n+1}), \ldots, T(s_{n+k}, s_{n+k+1}), P(s_n), \ldots, P(s_{n+k}) \Rightarrow P(s_{n+k+1})$
- Invariant:
  - satisfied by all reachable states of S



Proving Invariants by k-induction

- Automatable when
  1. I, T, P can be encoded in some formal logic $\mathcal{L}$
  2. Implication (=>) in $\mathcal{L}$ can be checked efficiently
- Challenges
  - Accuracy
    - encoding of T is often an over-approximation
    - k-induction is incomplete for proving invariant in infinite-state systems
  - Scalability
    - size of state space
    - size and complexity of resulting formulas

Improving Accuracy: Invariant Discovery

Problem

If P is invariant for S but k-induction fails to show it then $T(s, s')$ holds for some unreachable state s

Possible Solution

1. Find some other invariant $Inv(x)$ for S
2. Strengthen $T(x, y)$ into $T'(x, y) = T(x, y) \wedge Inv(x)$
3. Try again with T'

A Novel Invariant Discovery Method

1. Let $R(x,y)$ be a formula standing for a binary relation over program values (ex: $x => y$, $x = y$, $x < y$, $x - y < 10$, …)
2. Determine heuristically a finite set U of relevant terms (e.g., including selected terms from T)
3. Find a subset C of

   $$\{ (t_1, t_2) \text{ in } U \times U \mid R(t_1, t_2) \text{ is invariant } \}$$

4. Let $Inv = \wedge\{ R(t_1, t_2) \mid (t_1, t_2) \text{ in } C \}$

A Novel Invariant Discovery Method

1. Challenges:

   C should

   - be computed quickly
   - be fairly small
   - exclude pairs (t_1, t_2) with $R(t_1, t_2)$ valid

2.

3. Find a subset C of

   $$\{ (t_1, t_2) \text{ in } U \times U \mid R(t_1, t_2) \text{ is invariant } \}$$

4. Let $Inv = \wedge\{ R(t_1, t_2) \mid (t_1, t_2) \text{ in } C \}$

Initial Results

- Developed efficient method for computing set C when
  - R is over a basic data type (eg., bool, int, real)
  - R is reflexive, transitive and anti-symmetric (eg, =>, ≤, =, …)
- Properties of R allow a compact encoding(DAG) of C
- Method capitalizes on ability of SMT solvers to find counter-models

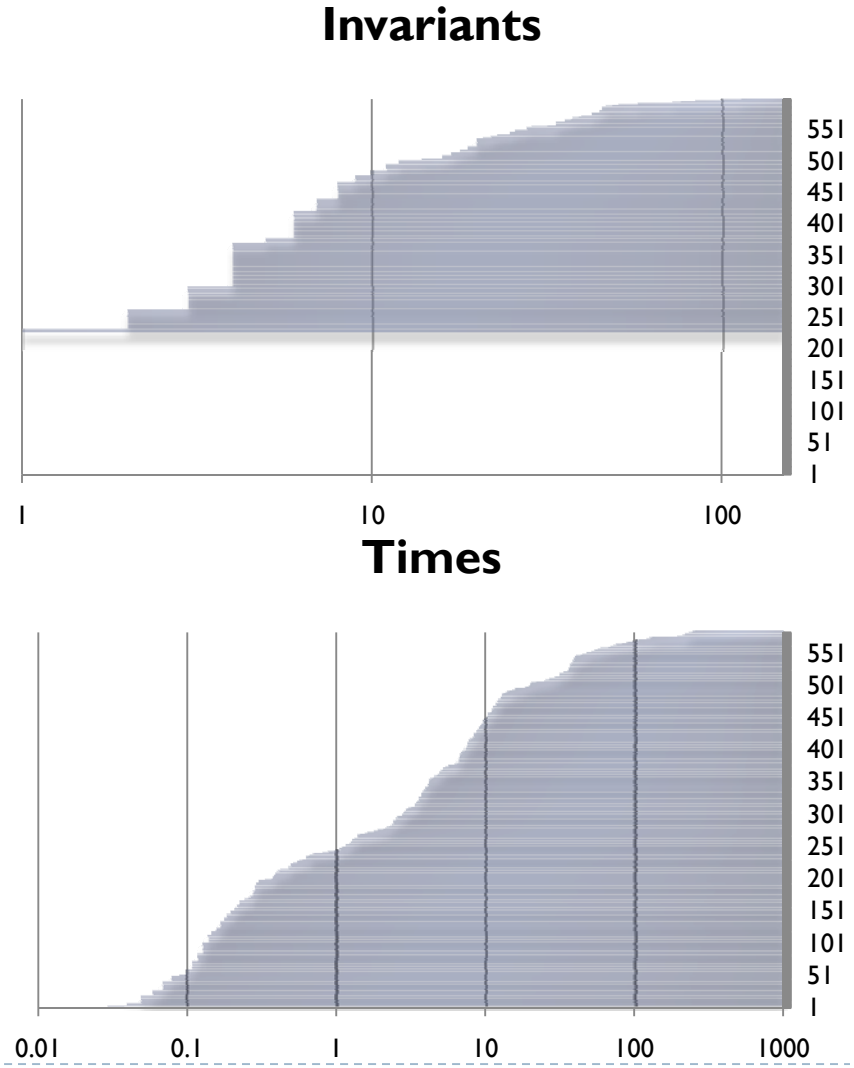
Initial Evaluation (=>)

- Considered ~600 Lustre benchmarks (program + property)
- All properties conjectured to hold
- Timeout: 120s / benchmark
- Kind with Yices SMT solver proves
  - 369 (61%) directly
  - 448 (75%) with the aid of separately discovered invariants
- Average proving times:
  - without invariants: 0.06s
  - with invariants: 0.29s

Invariant Discovery Statistics

- Timeout limit: 300s
- Timeouts: 17
- # of invariants/benchmark
  - Min: 0
  - Max: 168
  - Average: 8

- Generation time
  - Total: 130m
  - Average: 13s



Conclusion

- Automated verification of safety properties of synchronous data-flow systems specified in Lustre
- Translation of Lustre programs + properties into suitable FOL fragment with built-in theories
- Use of off-the-shelf SMT solvers to prove properties by *k*-induction methods
  - Enhanced with path compression, abstraction, & invariant discovery techniques
- Lustre verifier highly competitive with previous tools (Lesar, Luke, SAL)

Future Work

- Incorporation of in-house new SMT solver, CVC4, into Lustre verifier
- More powerful abstraction thanks to new CVC4 features
- Additional invariant discovery methods
- Modular verification
- Improved support for Lustre programs with non-linear arithmetic

Thank you