Efficient Generation of Set of Support for Safety Properties using UNSAT Cores and Induction *

Elaheh Ghassabani[1], Andrew Gacek[2], and Michael W. Whalen[1]

[1] Department of Computer Science and Engineering
University of Minnesota, 200 Union Street, Minneapolis, MN 55455,USA
{whalen,ghassaba}@cs.umn.edu
[2] Rockwell Collins Advanced Technology Center
400 Collins Rd. NE, Cedar Rapids, IA, 52498, USA
{andrew.gacek}@rockwellcollins.com

Abstract. Symbolic model checkers can construct proofs of properties over very complex models. However, the results reported by the tool when a proof succeeds do not generally provide much insight to the user. It is often useful for users to have traceability information related to the proof: which portions of the model were necessary to construct it. This traceability information can be used to diagnose a variety of modeling problems such as overconstrained axioms and underconstrained properties, and can also be used to measure *completeness* of a set of requirements over a model. In this paper, we present a new algorithm to efficiently compute the set of support within a model necessary for inductive proofs of safety properties for sequential systems. The algorithm is based on the UNSAT core support built into current SMT solvers and a novel encoding of the inductive problem to try to generate a minimal set of support. We prove our algorithm correct, and describe its implementation in the jkind model checker for Lustre models. We then present an experiment in which we benchmark the algorithm in terms of speed, robustness, and minimality, with promising results.

Keywords: Auto-traceability, Set of Support, Completeness, Requirement Engineering

1 Introduction

Symbolic model checking is an important verification technique for both hardware [] and software [**?**] systems, supporting a level of rigor beyond what is possible with testing. Current tools using induction-based techniques such as PDR [] and k-induction [] can often verify safety properties of complex infinite-state systems. However, in the event that a property is proved, it is not always clear what level of assurance should be invested in the result. Given that these kinds of analyses are performed for safety- and security-critical software, this can lead to overconfidence in the behavior of the fielded system. It is well known that issues such as vacuity [] can cause verification to succeed

* This work has been supported by XXX

despite errors in a property specification or in the model. Even for non-vacuous specifications, it is possible to over-constrain the specification of the *environment* in the model such that the implementation will not work in the actual operating environment.

Additionally, we are often interested in the *completeness* of requirements over a given implementation model. A model may satisfy its requirements, but these requirements may be sufficiently adequate to ensure the safe operation of the system. Although this problem is, at its root, domain dependent and slightly subjective, standards documents such as DO178C [] advocate measuring requirements completeness by measuring coverage of source code from tests derived from requirements. It is known [**?**] that this measure is indirect, and can only be performed late in the development cycle.

Finally, another aspect of the verification process that is important is that of traceability: it is necessary to map requirements to the locations in models or source code that lead to their satisfaction. Current traceability approaches involve either manual mappings between requirements and code/models [] or a heuristic approach involving natural language processing []. Both of these approaches tend to be inaccurate.

[Pivot to explanation of what we are doing]

- Overview of the problem: sequential model checkers do not provide much insight into proofs.
- Section should roughly follow the structure of "Finding Minimal Unsatisfiable Cores of Declarative Specifications" paper by Torlak et al (with Dan Jackson).
- UNSAT Cores have been used for a variety of analysis tasks
- we want to generalize this idea for sequential systems

2 Motivating Example

- Not sure if this should go before or after the background section with a description of Lustre.
- Need a small but interesting example. Andrew, do any of the models that you use as jkind tests function in this way? It would be nice to look at what we have lying around; we need something that requires invariants.
- It would also be good to have a few points of interest with the model-requirement pairing:
- vacuity due to an overconstrained environment
- definitions within the model that are irrelevant to the proof.
- Explain the model and the proof process.

3 Preliminaries

- Symbolic transition systems (use material from Sheeran's "Induction using a SAT Solver" paper?)
- Lustre language
- UNSAT cores
- jkind
- more here?

4 Set of Support

- What is it?
- How do we formalize it in terms of symbolic transition systems?
- How do we prove it correct?

5 Implementation

- This section will probably be fairly short.

6 Experiments

- Research questions (informally):
- RQ1: How much overhead does computation of set of support add to inductive proofs?
- RQ2: How stable is the computed set of support given different solvers and verification algorithms?
- RQ3: How close to minimal is the computed set of support compared to a much slower, but guaranteed minimal approach?
- Experimental setup: What did we use for benchmark models? Where did they come from? Why is this a good set?
- Tables related to each research question.

7 Related work

Alloy is a framework for describing high-level design of various systems, whose analyzer is a fully automatic constraint solver. Constraints are translated into propositional logic solved by a SAT solver; hence, the analysis considers only a finite number of values for each type. For this reason, even for a set of simple constraints, the analyzer is never able to prove the correctness of a property.

- MUS's
- Work on Alloy
- Work that Teme pointed us to.
- Anything else Elaheh has found.

8 Conclusions and Future Work

- Write this at the end.

Acknowledgments: We thank XXXX

References