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 using induction-based techniques such as PDR [**?**] and k-induction [**?**] can often determine whether safety properties hold of complex finite or infinite-state systems. Model checking tools are attractive both because they are automated, requiring little or no interaction with the user and, if the answer to a correctness query is negative, they provide a counterexample to the satisfaction of the property. These counterexamples can be used both to illustrate subtle errors in complex hardware and software designs [**?**, **?**] and to support automated test case generation [**?**, **?**]. In the event that a property is proved, however, it is not always clear what level of assurance should be invested in the result. Given that these kinds of analyses are performed for

* This work has been supported by XXX

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 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.

At issue is the level of feedback provided by the tool to the user. In most tools, when the answer to a correctness query is positive, no further information is provided. What we would like to provide is traceability information, e.g., a *set of support*, that explains the proof, in much the same way that a counterexample explains the negative result. This is not a new idea: UNSAT cores [**?**] provide the same kind of information for individual SAT or SMT queries, and this approach has been lifted to bounded analysis in [**?**]. What we propose is a generic and efficient mechanism for lifting the UNSAT core information into proofs of safety properties using inductive techniques such as PDR [**?**] and k-induction [**?**]. Because many properties are not natively inductively provable, these techniques introduce lemmas as part of the solving process in order to strengthen the property to make it inductively provable. Our technique allows efficient and accurate generation of sets of support to be extracted from the base and inductive verification steps in the presence of lemmas.

Once generated, the set of support information can be used for many purposes in the software verification process, including at least the following:

Vacuity detection: The idea of syntactic vacuity detection (checking whether all subformulae within a property are necessary for its satisfaction) has been well studied [**?**]. However, even if a property is not syntactically vacuous, it may not require substantial portions of the model. This in turn may indicate that either a.) the model is incorrectly constructed or b.) the property is weaker than expected. We have seen several examples of this mis-specification in our verification work, especially when variables computed by the model are used as part of antecedents to implications.

Completeness checking: Closely related to vacuity detection is the idea of *completeness checking*, e.g., are all atoms in the model necessary for at least one of the properties proven about the model? Several different notions of completeness checking have been proposed [**?**, **?**], but these are very expensive to compute, and in some cases, provide an overly strict answer (checking can only be performed on non-vacuous models for [**?**]).

Traceability: Certification standards for safety-critical systems (e.g., [**?**, **?**]) usually require *traceability matrices* that map high-level requirements to lower-level requirements and (eventually) leaf-level requirements to code or models. 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. The proof-based approach can provide this information accurately and for free.

Symbolic Simulation / Test Case Generation: Model checkers are now often used for symbolic simulation and structural-coverage-based test case generation [**?**, **?**]. For either of these purposes, the model checker is supposed to produce a witness trace for a given coverage obligation using a "trap property" which is expected to

be falsifiable. In systems of sufficient size, there is often "dead code" that cannot ever be reached. In this case, a proof of non-reachability is produced, and the set of support information provides the reason why this code is unreachable.

Nevertheless, to be useful for these tasks, the generation process must be efficient and the generated set-of-support must be accurate (that is, sound and close to minimal).

In the remainder of this paper, we present an algorithm for efficient generation of set-of-support information for induction-based model checkers. Our contributions, as detailed in the remainder of the paper, are as follows:

- We present a technique for extracting UNSAT cores from an inductive verification of a safety property over a sequential model involving lemmas.
- We formalize this technique and present an implementation of it in the jkind model checker [**?**]
- We present an experiment over our implementation and measure the efficiency, minimality, and robustness of the set-of-support generation process.

The rest of this article is organized as follows. In Section 2, we present a running example for the paper. In Section 3, we present the required background for our approach. In Sections 4 and 5, we present our approach and our implementation in jkind. Section 6 presents an evaluation of our approach on a set of benchmark examples. Finally, Section 7 discusses related work and Section 8 concludes.

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