Project 22703: SYRF

Deliverable 3.2.2

Use of the Stålmarck Method

---

1 Introduction

This deliverable presents the results achieved within task 3.2.2 during SYRF's second year of operation. At the previous review meeting, it was agreed that this task should be changed to reflect Prover Technology's strong interest in continuing to work in a synchronous setting, rather than extending to cover asynchronous systems as well. Prover has therefore developed LUCIFER, a tool for verifying LUSTRE programs, as well as continuing the work begun last year on the reduction of hardness in proofs by the addition of definitions of fresh variables. The work in each of these topics is described in the following two sections. Work on the LUCIFER tool has been partially funded by SYRF.

2 LUCIFER: a tool for verifying LUSTRE programs

To use Stålmarck's method to verify LUSTRE programs, one needs a translator from Lustre to propositional logic with arithmetic. Two prototype translators already existed at the beginning of 1998, one by Ågren (for SAO+ a commercial version of LUSTRE) [1] and one by Raymond from Verimag. Both of these produce flat formulas in which the hierarchical structure of the source program is lost. This has the disadvantage that it is hard to map information gained during the verification of the model produced by the translation (and in particular countermodels to required properties) back to the source program. This loses much of the advantage of the synchronous observers approach, in which the user keeps to the programming level, even when expressing the required properties of the system being designed and verified [7]. One wants feedback from the verification also at the programming level in order to ease debugging. It is for the failure case that the tool should be designed!

2.1 Preserving hierarchy

We have developed LUCIFER, a tool for verifying LUSTRE programs, in which the hierarchical structure of the source program is preserved in the model produced [10]. The model is a hierarchical structre of macros in the NP-Tools verification toolbox that implements Stålmarck's method (and which was demonstrated at the first review meeting). Macros provide a simple means of encapsulating sub-parts of a formula, rather like a hierarchical circuit description.

The macro structure of the model produced by LUCIFER mirrors exactly the node structure of the LUSTRE source program, even down to the variable names. The model is a formula in propositional logic with integer arithmetic expressing the initialisation and the transition function of the LUSTRE program, which is assumed to include a synchronous observer expressing a safety property. (Note the restriction to integer arithmetic. We discuss the effects of this and the subset of Lustre currently supported in a later section. There are various ways in which we could extend LUCIFER to deal with real variables and we would like to consider this extension during the coming year.)

2.2 Proofs by induction

Proofs always check that a single output signal (called OK) is always true, and they are by induction. Normally, NP-Tools is used to prove the base case, that the OK signal is true at time zero, and that if it is true at some arbitrary time n, then it is also true at time $n+1$. That is, we analyse a single time unit in the base case, and a single copy of the transition function in the step. Sometimes though, depending on the shape of the program to be analysed (in terms of its configuration of `pre` operators), it is necessary to consider instead k initial steps and k transitions from an arbitrary state. These kinds of inductive verification proofs are used routinely by Prover Technology in industrial projects. So far, we have found that k is actually quite small in real examples. However, we now feel the need to do further theoretical work on this form of program verification. It would be good to have a better understanding of the class of programs for which it is appropriate, and of its relationship to bounded model checking, which we discuss briefly below. We hope also to benefit from the expertise of our SYRF partners in syncronous program verification. Now that we have a working translator, it will be a lot easier to have an active collaboration in this area.

At present, the translator takes a positive k as a parameter. We have developed a rather naive program analysis that finds a lower bound on k by finding the length (in terms of `pre` operators) of the longest path from input to output. A more effective program analysis method would be one of the results of further theoretical work in this area.

2.3 The subset of LUSTRE supported by the current prototype

The current LUCIFER prototype works on a restricted subset of version 3+ of LUSTRE [12]. As mentioned above, we have no means of dealing with real variables and this is a restriction that we would like to remove. To ensure that the model in NP-Tools is sound, all variables of type real are ignored. However, use of the equality and comparison operators on reals still produces (free) boolean values. For similar reasons, we are unable to deal with variables of imported types.

The other main restriction is on the use of the temporal operators `current` and `when`. Only two kinds of expressions containing these operators are allowed:

$$\begin{aligned} V_1 &= \texttt{current}(E\ \texttt{when}\ B); \\ V_2 &= \texttt{current}(f_1(f_2(\ldots(f_n(E\ \texttt{when}\ B))))); \end{aligned}$$

where E is a general expression and B is a boolean expression. Neither E nor B is allowed to contain `when` or `current` operations. The nodes f_1, f_2, $\ldots f_n$ are general nodes, with the draconian limitation that they only take one argument. However, it is always possible to place multiple arguments to the function into a tuple.

2.4 Case studies

2.4.1 The lift controller

The lift controller from [8] has been analysed. Of the six properties proved by Holenderski, five are proved instantly in NP-Tools from the model generated by LUCIFER. The sixth gives a false countermodel because the original LUSTRE code uses imported functions, whose outputs are translated to free variables and then assigned unrealistic values in the countermodel. An adaption of the program that implements the imported functions, but also fixes the number of floors, gives instant verification of all six properties. In all cases, the induction base required was two.

2.4.2 A small railway system

As part of the EU project CRISYS [5], Caspi and Salem have given a case study of a small railway system [3] The system is a control system for a single line railway and a synchronous observer expresses the property that there cannot be any collisions or derailments of trains. The specification consists of three parts: the physical part (tracks, trains, sensors, actuators etc.), the control program in LUSTRE, and the humans controlling the trains, which are assumed to behave according to certain rules. Both the physical part and the humans are modelled as part of the synchronous observer as a number of `assert`s. The required safety properties are proved instantly in NP-Tools, again with an induction base of two.

2.4.3 Cruise controller

Part of a cruise controller written in the specification language TRIO was manually translated to LUSTRE and verified using LUCIFER. In this case, the required induction base was three. LUCIFER was found to be particularly helpful during the writing of the LUSTRE specification. It was especially useful to be able to see how the values inside the transition function were propagated when the requirements were falsifiable. It seems likely that LUCIFER's role in the design cycle will be to support program construction in this way, rather than as a *post hoc* verification tool.

2.4.4 Airbag control

In a commercial project, the logic controlling airbag release in a car was verified. Some strange behaviour was detected in a verification requiring over a hunderd of copies of the transition function. The success of the project bodes well for the viability of LUCIFER as a commercial product.

2.4.5 Schneider case study

We have recently begun work on the Schneider case study. Our aim is to compare results from using LUCIFER with those from LESAR and LURETTE.

2.5 Incorporating bounded model checking

In a recent development, Ed Clarke and his group have introduced Bounded Model Checking (BMC), a variant of model checking in which only sequences of states of bounded length are considered. (The method was presented in an extra talk at the recent International Conference on Formal Methods in CAD, Palo Alto, November 1998.) Clarke et al have shown how to reduce unbounded model checking to bounded model checking, though as in our inductive proofs, it is generally very hard to find the necessary bound. They have also shown how bounded model checking can be coded in such a way that it reduces to boolean satisfiability. In a number of case studies, Clarke et al have shown that using Stålmarck's method for boolean satisfiability gives remarkably good results when compared to standard BDD-based model checking. This seems to be a major development in the field of model checking. Therefore, we are working on incorporating BMC into LUCIFER, although we had not originally planned to incorporate model checking. The necessary coding is rather close to the unrolled transition functions that we already produce.

2.6 Future work on LUCIFER

From a practical point of view, it is important to introduce support for variables of type real. It would also be useful to support imported types, perhaps by means of some sort of theory mechanism. It is also important to lift the restriction on the use of the `current` and `when` operators.

We plan to investigate test pattern generation in the context of LUCIFER. Work on test pattern generation for digital circuits using recursive learning (which is very similar to Stålmarck's method) has given promising results [9].

A graphical user interface that directly supports the main tasks of the LUCIFER user would be desirable. Activities such as choice of induction base and exploration of countermodels are tedious for the user at present. A good integration with other LUSTRE compilation and analysis tools would also increase the usefulness of our tool.

It would be very interesting to translate LUSTRE to many sorted first order logic (msfol) and experiment with Lundgren's prototype implementation of Stålmarck's extension of his method to msfol [11].

3 Reducing hardness of proofs in Stålmarck's method

3.1 Background

This work continues that demonstrated at the review meeting for year 1 of SYRF. According to Stålmarck, a formula is k-easy if it has a proof in the dilemma proof system that has at most k simultaneous assumptions, that is has depth at most k. If there is no shallower proof, the formula has hardness degree k [16]. The k-saturation procedure implemented in verification tools based on Stålmarck's method searches exhaustively for a proof of depth k. Thus it refutes a contradictory formula of hardness degree k by finding a short proof that leads to a contradiction. The dilemma proof system, the notion of hardness and the k-saturation algorithm are all explained in a new tutorial on Stålmarck's method that is an annex to deliverable 0.2.2 [15].

In work done during 1997, we showed how to reduce the degree of hardness of proofs in Stålmarck's method by adding carefully chosen definitions of fresh variables to the system description. The extra definitions, which we call helper variables, increase the set of subformulas available to the prover and can reduce branching and so proof complexity. In some cases, the extra subformulas are branched on in the new (shallower) proof, but in other cases they simply aid the propagation of information via the simple rules. The best way to understand these helper variables is to think of doing a pencil and paper proof by making a clever definition and then using it to give a shorter more elegant proof than would otherwise have been possible. Choosing the right helper variables gives a formula that is larger but has lower hardness degree than the original.

We produce formulas and additional definitions from high level descriptions written in Haskell using the Lava system. Lava embeds a simple synchronous hardware description language in Haskell, and analyses circuit descriptions in various ways using symbolic evaluation. One of these analyses produces propsitional formulas for input to an implementation of Stålmarck's method. A paper describing the Lava system and its use in formal verification of circuits was published this year [2]. It is included as an annex to this deliverable.

During 1997, we concentrated on verification of regular combinational arithmetic circuits *across levels of abstraction*. For example, we verified that a bit-level adder, together with conversions from integer inputs into bit-vectors and from a bit-vector to an output integer, actually performed integer addition. This work has now been published [14] and appears as an annex to this deliverable. The kind of compositional reasoning used here will probably be vital in the verification of large systems. However, as in other work with similar aims (for example Bryant's use of *BMDs), it demands either that the user knows the shape of the proof and provides the necessary abstraction functions or that the shape of the proof can be deduced from the shape of the program or circuit. Nevertheless, we expect that hierarchical formal verification will become increasingly important, and we plan to return to this topic.

3.2 Hardness reduction in purely propositional proofs

In 1998, though, we decided to investigate hardness reduction in purely propositional proofs. Equivalence checking of bit-level components is an important building block of any verification system. The motivating problem is the comparison of bit-level multipliers. The degree of hardness of the resulting formulas increases with circuit size. If we can find out how to verify regular circuits at the bit level, we should be able to use that knowledge to ease proof hardness for the inductive proofs used in LUCIFER. A formula that connects several copies of a transition function is also a sort of regular circuit. And we cannot use abstract integer values to reduce the proofs to a sequence of identical steps as we did before. The problem has turned out to be harder than we thought! We have not yet achieved our goal of reducing proof hardness for multiplier comparison to a constant (preferably one).

3.3 Pigeon hole formulas and multipliers

After many fruitless experiments, we retreated to a more abstract setting in which the formulas of a given hardness are smaller and so produce proof logs that are easier to study (though still very large). We considered the classic pigeon hole formulas, which encode the assertion that there is no one to one function from a set of size n to a set of size $n+1$ [4]. This gives a sequence of propositional formulas for $n=1$, $n=2$ and so on, and the degree of hardness of the proofs that these are tautologies increases with n. We had earlier succeeded in reducing proof hardness (for a fixed n) by one or two by adding helper variables corresponding to smaller pigeon hole formulas. This year, we have managed to find a recursive reformulation of the pigeon hole formulas that adds sufficient new subformulas to reduce the hardness degree to a constant one independent of the size of the instance of the pigeon hole formula. This has greatly increased our understanding of how to 'force' the proof to follow the shape of the formula in a pattern that mimics the kind of inductive proof that a human might do.

We are now working on transferring the knowledge gained on these abstract formulas back to the problem of how to prove regular circuits at the bit level. We have managed to reduce hardness for multiplier comparison by a constant amount, but not yet *to* a constant. In particular, we have found that a single step, in which we show that combining an adder with a circuit that is asserted to be a multiplier again gives a multipler, has itself increasing hardness degree with increasing circuit size. This has prompted us to work first on reducing the hardness of adder comparison from two to one. (Some of the adders studied were obtained from SYRF partners.) We have succeeded in this, for example by doing a separate proof of the equivalence of each pair of outputs from the adders being compared rather than a monolithic proof of a large conjunction of equivalences. We are now working on reducing the hardness of the step, again by the addition of smaller recursive subformulas. Part of this work is being done by Jonas Eden as a Master's project at Prover Technology [6].

3.4 Future work on hardness reduction

We still expect to succeed in reducing the hardness of bit-level multiplier comparison to a constant (hopefully one). However, the difficulties that we have encountered perhaps indicate that it would be wise to return to work on verification across levels of abstraction, where one can choose to express the problem in such a way that it is easy to guide the proof using helper variables.

What we really need is a calculus of proof hardness that can be used to help in choosing strategies for hardness reduction. This idea was suggested by Koen Claessen (a doctoral student of Sheeran's at Chalmers) and he is now working on this topic. The experimental work that we have done so far has greatly increased our understanding of how Stålmarck's method works in practice on large formulas. However, it is now time to combine this practical experience with a more careful theoretical analysis.

We are also studying Richards' suggestions for enhancements to Stålmarck's method [13], and plan to experiment with having simple rules about larger operators (or function blocks) than just the standard boolean connectives.

References

[1] H. Ågren: SAO+/FD to NPC – Language Compatibility and Translatability. Technical Report U-96042, Logikkonsult NP AB, 1996.

[2] P. Bjesse, K. Claessen, M. Sheeran and S. Singh: Lava: Hardware Design in Haskell. Proc. Int. Conf. on Functional Programming, ACM Press, 1998.

[3] P. Caspi and R. Salem: A Synchronous Case Study (Appendix to Deliverable D11.1). Technical Report, Esprit Project CRISYS 25514, 1998.

[4] S.A. Cook: The complexity of theorem-proving procedures. In Proc. 3rd ACM Symp. on the Theory of Computing, 1971.

[5] CRISYS – CRitical Information and control SYStem, 1997. Esprit RTD project 25514.

[6] J. Eden: Reducing proof hardness in propositional logic (draft), report on Master's Thesis work done at Prover Technology, for a degree at Chalmers University of Technology, Sweden, 1998.

[7] N. Halbwachs, F. Lagnier and P. Raymond: Synchronous observers and the verification of reactive systems. In Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Springer Workshops in Computeting, 1993.

[8] L. Holenderski: A Lift controller in Lustre. In R.J.R. Back and K. Sere, editors, Proc. of the 5th Nordic Workshop on Program Correctness, 1994.

[9] W. Kunz and D.K. Pradhan: Recursive Learning: A New Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization. IEEE Trans. CAD, vol. 13, no. 9, 1994.

[10] Formal modelling and automatic verification of Lustre programs using NP-Tools (draft), report on Master's Thesis work done at Prover Technology, for a degree at Royal Institute of Technology, Sweden, 1998.

[11] Stålmarck's method in many-sorted first order logic (draft), report on Master's thesis work done at Prover Technology, for a degree at Chalmers University of Technology, Sweden, 1998.

[12] *Lustre language reference manual, Versions 3, 3+, 4, 5*, September 1997.

[13] M. Richards: A tautology Checker loosely related to Stålmarck's algorithm, slides of a talk given at Cambridge University Computer Laboratory (available from www.cl.cam.ac.uk/users/mr/), March 1998.

[14] M.Sheeran and A. Borälv: How to prove properties of recursively defined circuits using Stålmarck's method. Proc. Workshop on Formal Methods for Hardware and Hardware-like systems, Marstrand, June 1998.

[15] M. Sheeran and G. Stålmarck: A tutorial on Stålmarck's proof procedure for propositional logic. In Proc. Int. Conf. on Formal Methods in Computer Aided Design (FMCAD'98), Springer LNCS, 1998.

[16] G. Stålmarck: A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula, 1989. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (approved 1994), European Patent No. 0403 454 (approved 1995).