Suraq — A Controller Synthesis Tool using Uninterpreted Functions*

Georg Hofferek[1] and Ashutosh Gupta[2]

[1]Graz University of Technology, Austria
[2]IST Austria

Abstract. Boolean controllers for systems with complex datapaths are often very difficult to implement correctly, in particular when concurrency is involved. Yet, in many instances it is easy to formally specify correctness. For example, the specification for the controller of a pipelined processor only has to state that the pipelined processor gives the same results as a non-pipelined reference design. This makes such controllers a good target for automated synthesis. However, an efficient abstraction for the complex datapath elements is needed, as a bit-precise description is often infeasible. We present SURAQ, the first controller synthesis tool which uses uninterpreted functions for the abstraction. Quantified first-order formulas (with specific quantifier structure) serve as the specification language from which SURAQ synthesizes Boolean controllers. SURAQ transforms the specification into an unsatisfiable SMT formula, and uses Craig interpolation to compute its results. Using SURAQ, we were able to synthesize a controller (consisting of two Boolean signals) for a five-stage pipelined DLX processor in roughly one hour and 15 minutes.

1 Introduction

When developing a complex digital system, some parts are more difficult to implement correctly than others. For example, creating a combinational circuit that multiplies two 64-bit integers is easier than implementing the stall and forwarding logic of a pipelined microprocessor. On the other hand, some system parts are also easier to formally specify than others. For the pipeline controller, the specification simply states that the execution of any program on the pipelined processor should output the same results as executing the same program on a non-pipelined reference processor. This notion has been introduced by Burch and Dill [5], who used this paradigm for verification of pipelined processors. Another key feature of their work was the use of uninterpreted functions for abstraction of complex datapath elements. A bit-precise description of, e.g., a multiplier would have been exponentially — and thus prohibitively — large. Hofferek and Bloem [12] have shown how to turn this verification setting into a synthesis setting. They introduced specifications that are quantified first-order formulas

* The work presented in this paper was supported in part by the European Research Council (ERC) under grant agreement 267989 (QUAREM) and the Austrian Science Fund (FWF) through projects RiSE (S11406-N23) and QUAINT (I774-N23).

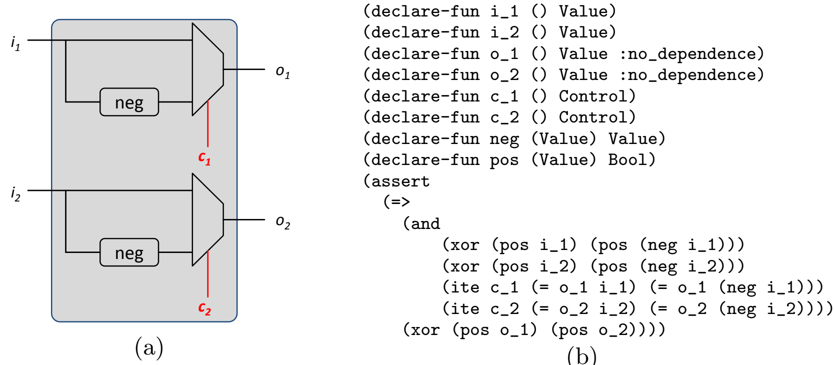

Fig. 1. (a) A controller synthesis example with missing controllers of signals c_1 and c_2. The specification for the controllers states that the outputs always have opposite signs. (b) The corresponding sythesis query to SURAQ in SMTLIB-like format.

which state that *for all inputs/states, there exist values for Boolean control signals such that (for all values of auxiliary variables) a correctness criterion Φ holds.* The formula Φ can be a Burch-Dill style verification condition, or — for different applications — another first-order formula that states correctness of the system in question. The *certificates* for the existentially quantified Boolean control signals is a correct-by-construction implementation of the controller. One way to compute such certificates — which is based on (a generalization of) Craig interpolation [6] — has been introduced in [13].

In this paper, we present SURAQ [1], an open source tool that implements the synthesis approach of [13]. The most impressive result we achieved with SURAQ so far is the synthesis of two Boolean control signals for a five-stage pipelined DLX processor [10]. The required time for this synthesis is roughly one hour and 15 minutes. More details on SURAQ can be found in [11].

Related Work. Research on automated synthesis has flourished over the last years. A lot of work (e.g. [14, 17, 7, 18, 16, 8]) is concerned with *synthesis of reactive systems from temporal specifications*. However, the specification languages used by these approaches are bit-precise. Thus, they are not suitable for the controller synthesis problems we consider. Our approach is closer to *program sketching* [19], a mixed imperative/declarative paradigm where parts of a program are implemented manually and some missing (constant) expressions are synthesized automatically. *Functional synthesis* [15] by Kuncak et al. is orthogonal to our work. Whereas we assume that data operations are already implemented, they focus on synthesizing data-oriented functions from specifications about their input-output behavior.

2 Synthesis Method

SURAQ implements the synthesis method presented in [13], although with some improvements. We start from a formula of the form

$$\forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi, \qquad (1)$$

where $\overline{c}$ is a vector of Boolean control signals which we want to synthesize, $\overline{x}$ and $\overline{x}'$ are vectors of first-order variables, and Φ is a formula in the combination of the quantifier-free fragment of the theory of uninterpreted functions and equality ("`QF_UF`"), and the array property fragment [4]. A precise definition of this combination of theories is given in [12]. SURAQ first performs the *index set construction* [4] to reduce Φ to an equivalent formula in `QF_UF`. Next, SURAQ instantiates the existential quantifier, renames the universally quantified $\overline{x}'$ variables in each of the resulting $2^{|\overline{c}|}$ instantiations, and negates the whole formula. This yields an unsatisfiable SMT formula in `QF_UF`. SURAQ uses the VERIT SMT solver [3] to obtain a refutation proof.

Based on this refutation proof, SURAQ supports two modes. In *iterative mode*, SURAQ first computes a solution for one control signal, using the interpolation method of Fuchs et al. [9]. This solution is then resubstituted into the original formula, before performing the aforementioned reduction, expansion (now yielding only $2^{|\overline{c}|-1}$ instantiations), and transformation again. From the resulting SMT instance, the solution for the next control signal is computed. This is repeated until solutions for all control signals have been obtained.

In contrast to this, in *n-interpolation mode*, SURAQ computes all control signals from the first refutation proof. To perform this so-called n-interpolation, the proof must be made *colorable* and *local-first* [13]. To obtain these properties, we follow the proof transformations outlined in [13], with one significant improvement: We do not perform the transformation to remove non-colorable literals from the proof. Instead, when parsing the proof, we immediately discard the subproofs of any proof nodes that are solely derived from theory lemmata. This way, the proofs never contain any non-colorable literals. Splitting of non-colorable theory lemmata is done in parallel. SURAQ provides a command-line parameter to specify how many threads should be used for splitting.

2.1 Using Suraq

As an input, SURAQ requires a specification in form of a formula as shown in Equation 1. The formula Φ should be given in SMTLIB-like [2] format. The quantifier prefix is implicitly given by the variable declarations. Variables declared with sort `Control` are bound by the existential quantifier (and thus, certificates for them should be synthesized). Variables declared with a `:no_dependence` attribute are bound by the inner universal quantifier. Thus, these are auxiliary variables that the synthesized functions cannot depend on. All other variables are bound by the outer universal quantifier. An example is shown in Figure 1.

As its output, SURAQ also produces a file in SMTLIB format, where the solution for each control signal is given as an expression of the form `(assert (= c_i <expr.>))`. Moreover, the declarations and main formula from the input file is copied, in a slightly modified way: The sort `Control` is replaced by `Boolean`, all `:no_dependence` attributes are removed, and the main formula is negated. This way, the output file can directly be used for third-party verification of the synthesis result. One simply has to give the file to an SMT solver, which will return `unsat` if the result is correct.

Table 1. Runtime Results (n-Interpolation Mode). Column 1 names the benchmark. Column 2 gives the time for the formula reductions, that is, the total time required for reading the specification, performing the formula reductions, and creating an input file for veriT. Column 3 gives the time required by veriT to produce a proof. Column 4 gives the (wall clock) time taken to split all non-colorable theory lemmata, using 24 parallel threads. Column 5 gives the time taken by veriT for propositional SAT solving with the stronger theory lemmata obtained from splitting. Column 6 gives the time for reorder the proof to make it local-first. Column 7 gives the time spent on proof parsing, including splitting of multi-resolution nodes. This combines the time for parsing the SMT proof and the propositional SAT proof. Column 8 gives the total time of synthesis. All times are given in seconds, and rounded to integers.

1	2	3	4	5	6	7	8
Name	Formula Reduction	SMT Solving	Splitting Leaves	SAT Solving	Re-ordering	Proof Parsing	Total
simple_pipeline	<1	<1	<1	<1	<1	<1	1
illus_02	<1	<1	<1	<1	<1	<1	<1
illus_03	<1	<1	<1	<1	<1	<1	1
illus_04	1	<1	<1	<1	<1	<1	2
illus_05	2	<1	<1	<1	<1	<1	3
illus_06	4	<1	<1	<1	<1	<1	5
illus_07	7	<1	<1	<1	<1	<1	11
illus_08	14	1	<1	<1	<1	<1	17
illus_09	28	3	<1	<1	<1	<1	34
simple_processor	<1	<1	<1	<1	<1	<1	4
dlx_stall_f-a-ex	6	1718	6	7	n/a	442	n/a

3 Experimental Results

We have evaluated Suraq with several benchmarks. First, we used the simple pipeline example from [12]. Furthermore, we used several instances of the scalable, illustrative example from [13] (see also Fig. 1). We also tried the simple, two-stage pipelined processor from [13]. Finally, to demonstrate the applicability of our approach to real-world problems, we synthesized a controller for a five-stage pipelined DLX processor [10]. We have created several variants of the DLX benchmark, where we synthesize different control signals (while the other are implemented manually); in the dlx_stall_f-a-ex benchmark, we even synthesize 2 signals simultaneously.

In Table 1, we present runtime results for n-interpolation mode. Note that the reordering of the resolution proof times out for the dlx_stall_f-a-ex benchmark. In Table 2, we give sizes of the proofs (in various stages of transformation). Table 3 gives results (runtimes and proof sizes) for the iterative mode.

From this data, we can see that neither iterative mode, nor n-interpolation mode is clearly superior over the other. Instead, it depends on the characteristics of the benchmark which approach performs better. While for some benchmarks n-interpolation clearly outperforms iterative interpolation, in other instances the need for proof reordering makes n-interpolation inapplicable.

Table 2. Proof Sizes. The Col. 1 gives the name of the benchmark. Col. 2 states the size of the proof, as obtained from veriT, however with subproofs of theory lemmata already removed. Col. 3 gives the number of leaves that are non-colorable and need to be split, and Col. 4 gives the total number of leaves. Col. 5 gives the size of the proof obtained by calling a SAT solver on the skeleton of the original formula, together with the colorable theory lemmata and the (stronger) theory lemmata obtained from splitting. This is the proof that is given to the reordering procedure. The size of the proof after reordering is given in Col. 6. Col. 7 gives the size of the proof that is used for n-interpolation, that is, the reordered proof with local subproofs removed. All proof sizes are given as the number of nodes in the DAG.

1	2	3	4	5	6	7
Name	Original Proof	# Leaves to split	# Leaves (total)	Before Reordering	After Reordering	w/o Local Subproofs
simple_pipeline	506	2	178	496	494	12
illus_02	102	2	44	106	106	12
illus_03	179	3	77	198	218	26
illus_04	390	7	133	356	428	46
illus_05	408	9	165	700	971	115
illus_06	669	4	176	758	1 576	320
illus_07	1 006	11	219	916	2 823	785
illus_08	1 101	6	242	2 214	8 082	1 347
illus_09	1 101	7	269	1 388	5 364	1 293
simple_processor	9 576	123	1 503	6 853	7 899	73
dlx_stall_f-a-ex	856 121	2 748	21 349	333 260	n/a	n/a

4 Conclusion

Suraq is a controller synthesis tool based on the method presented in [13]. Suraq has successfully synthesized a controller for a five-stage pipelined DLX processor [10]. Since the DLX benchmark is of realistic size and complexity, our experiments suggest that the approach is scalable enough for real-world problems.

References

1. Suraq — Synthesizer using Uninterpreted functions, aRrays and eQuality. `http://www.iaik.tugraz.at/content/research/design_verification/suraq/` (2014)
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Proc. of the 8th Int. Workshop on Satisfiability Modulo Theories (2010)
3. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: CADE (2009)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI (2006)
5. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: CAV (1994)
6. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. The Journal of Symbolic Logic 22(3), pp. 269–285 (1957)

Table 3. Iterative Mode. The 8 columns after the name give the SMT solving time (in seconds) and the proof size per iteration, in the format "time; size". Columns not required are left empty. The last column gives the total synthesis time.

	Iteration									Total
Name	1	2	3	4	5	6	7	8	9	Time
simple_pipeline	<1; 506									<1
illus_02	<1; 102	<1; 166								1
illus_03	<1; 179	<1; 493	<1; 508							2
illus_04	<1; 390	<1; 680	<1; 724	<1; 1 251						3
illus_05	<1; 408	<1; 2 133	<1; 3 608	<1; 3 298	<1; 3 361					6
illus_06	<1; 669	<1; 2 521	<1; 1 799	<1; 3 906	<1; 9 043	<1; 10 088				12
illus_07	<1; 1 006	<1; 6 430	<1; 7 210	1; 26 072	<1; 23 941	<1; 26 543	<1; 32 009			31
illus_08	1; 1 101	<1; 7 352	<1; 3 332	1; 16 312	2; 32 087	2; 52 782	2; 60 822	1; 73 887		66
illus_09	3; 1 101	5; 27 210	22; 60 002	45; 165 636	24; 117 535	23; 243 332	10; 231 789	9; 391 277	6; 281 313	485
simple_processor	<1; 9 576	<1; 8 682								4
dlx_stall	267; 898 345									537
dlx_f-a-ex	573; 1 490 028									1 358
dlx_f-b-wb	590; 2 271 288									2 174
dlx_stall_f-a-ex	1711; 856 121	923; 1 460 582								4 528

7. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: CAV (2009)
8. Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: VMCAI (2012)
9. Fuchs, A., Goel, A., Grundy, J., Krstic, S., Tinelli, C.: Ground interpolation for the theory of equality. Logical Methods in Computer Science 8(1) (2012)
10. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 2nd Edition. Morgan Kaufmann (1996)
11. Hofferek, G.: Controller Synthesis with Uninterpreted Functions. Ph.D. thesis, Graz University of Technology (July 2014)
12. Hofferek, G., Bloem, R.: Controller synthesis for pipelined circuits using uninterpreted functions. In: MEMOCODE (2011)
13. Hofferek, G., Gupta, A., Könighofer, B., Jiang, J., Bloem, R.: Synthesizing multiple boolean functions using interpolation on a single proof. In: FMCAD (2013)
14. Jobstmann, B., Galler, S.J., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In: CAV (2007)
15. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT 15(5-6) (2013)
16. Morgenstern, A., Schneider, K.: Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis. In: GANDALF (2010)
17. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: ATVA (2007)
18. Sohail, S., Somenzi, F.: Safety first: a two-stage algorithm for the synthesis of reactive systems. STTT 15(5-6) (2013)
19. Solar-Lezama, A.: Program sketching. STTT 15(5-6) (2013)