Software Synthesis is Hard – and Simple

Sven Schewe
University of Liverpool
sven.schewe@liverpool.ac.uk

Abstract

While the components of distributed hardware systems can reasonably be assumed to be synchronised, this is not the case for the components of distributed software systems. This has a strong impact on the class of synthesis problems for which decision procedures exist: While there is a rich family of distributed systems, including pipelines, chains, and rings, for which the realisability and synthesis problem is decidable if the system components are composed synchronously, it is well known that the asynchronous synthesis problem is only decidable for monolithic systems. From a theoretical point of view, this renders distributed software synthesis undecidable, and one is tempted to conclude that synthesis of asynchronous systems, and hence of software, is much harder than the synthesis of synchronous systems. Taking a more practical approach, however, reveals that bounded synthesis, one of the most promising synthesis techniques, can easily be extended to asynchronous systems. This merits the hope that the promising results from bounded synthesis will carry over to asynchronous systems as well.

1 Synthesis

In synthesis, we try to automatically construct a system from its formal specification [5, 22, 7, 15, 18]. If synthesis fails, the unrealisability of the specification demonstrates an error in the specification, or at least the incompatibility of a partially completed design with its specification [9]. In *software synthesis*, we would assume the system components to be composed *asynchronously*, a problem that has enjoyed far less attention than the problem of synthesising systems of synchronised components.

Church's solvability problem [5] can be identified as the origin of distributed (synchronous/hardware) synthesis. In 1962, Church [5] raised the question whether we can, for a given a relation $R \subseteq (2^I)^\omega \times (2^O)^\omega$ in the monadic second order logic of one successor (S1S), decide if there is a function $p : (2^I)^\omega \to (2^O)^\omega$ such that $(\pi, p(\pi)) \in R$ satisfies the relation for all infinite sequences $\pi \in (2^I)^\omega$.

In his solvability problem, Church distinguishes the input variables of a module, which are not under its control, from its output. He thus introduces the notion of a predefined interface between a module and its environment. Church's solvability problem triggered several deep results, including Büchi and Landweber's studies on finite games of infinite duration [4, 3] and Rabin's works on finite automata over infinite structures [21, 22].

Dagstuhl Seminar Proceedings 09501
Software Synthesis
http://drops.dagstuhl.de/opus/volltexte/2010/2670

While inspired by Church's problem, these automata and game theoretical results have found their application in *model checking*, the simpler problem of checking if an implementation is a model of its specification. The success of model checking has been preceded by a significant simplification of the specification languages from monadic second order logic to temporal logics like LTL [17], CTL [7], or, more recently, ATL [1]. Its success has been driven by the development of several tools like SPIN [11], MOCHA [2], and NuSMV [6], which can automatically analyse medium to large sized verification problems.

A long term research goal is the development of comparable support tools for the *construction* of reactive systems. Turning towards the harder task of synthesis, we will face the problem of accounting for the *incomplete information* [24, 12, 25] that the local interfaces of components reveal from the global system state. In model checking, incomplete information has no effect whatsoever, as it does not matter if a module does not react on an event because it does not see it (and hence cannot react), or because it does not choose to. In synthesis, however, the restricted access to information must be taken into account by the synthesis algorithm.

The generalisation of Church's solvability problem to a distributed setting [20, 14, 13, 9] is equivalent to solving a multi player zero sum game [27]. Pnueli and Rosner [20] showed that the problems occurring in distributed synthesis resemble those known from peek games with incomplete information [16, 23], and thus give Turing power even to simple specification languages such as LTL [20] or CTL [9]. However, important classes of systems, such as pipelines [20], chains, and rings [13], account for a hierarchy in the informedness of processes. For such systems, synthesis is decidable [9, 20, 13], albeit with high complexity.

For systems that are composed asynchronously, any reasonable scheduling mechanism will destroy such an order. Consequently, all architectures but monolithic ones come with an undecidable synthesis problem [25].

2 Bounded Synthesis

The high complexity of distributed synthesis has lead to an argument against the feasibility of distributed synthesis, in particular compared to model checking. However, we argue with Kupferman and Vardi [13] that this comparison is misleading, because the high complexity of distributed synthesis is caused by equally high lower bounds on the maximal size of a minimal model, or, outside of the decidable fragment, by the lack of such lower bounds. Hence, when comparing the input complexity, the size of some (not necessarily minimal) model has already entered in case of model checking, while synthesis algorithms are supposed to take the blame for the incurred blow-up.

As a consequence of this observation, Schewe and Finkbeiner developed the concept of *bounded synthesis* [26], where the search space is restricted to systems whose size does not exceed a predefined bound. This results in a shift from input complexity to output complexity, and levels the playing field for the synthesis vs. model checking comparison. This is particularly interesting for distributed synthesis: Once we have fixed a bound on the size of the systems we are interested in, we can reduce the synthesis problem to a simple Satisfiability Modulo Theories (SMT) problem [10]. Bounded synthesis has later been reinvented by Filiot et al. [8], who found further evidence of its practical applicability.

3 Asynchronous Bounded Synthesis

The theoretical argument against the feasibility of asynchronous synthesis is even stronger than the argument against synchronous synthesis, but, fortunately, the counter-argument remains valid: Using the specification only as input for the synthesis problem and the specification *plus* the—usually much larger—implementation as input for model checking naturally leads to an unfair advantage of model checking over synthesis.

Bounded synthesis, however, seems to be a silver bullet: The application of SMT techniques is based on guessing a minimal implementation and its composition; in order to extend these techniques to software synthesis, it suffices to change the composition rules from those for synchronous composition (as currently used in [26, 10]) to those for asynchronous compositions, for example by using the composition rules described in [25].

This way, we inherit the advantage of bounded synthesis: We shift from an infeasible input complexity to a low complexity in the size of the minimal system, levelling the playing field between model checking and synthesis again.

References

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.

[2] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV 1998)*, pages 521–525, 1998.

[3] J. R. Büchi and L. H. Landweber. Definability in the monadic second-order theory of successor. *Journal of Symbolic Logic*, 34(2):166–170, 1969.

[4] J. R. Büchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[5] A. Church. Logic, arithmetic and automata. In *Proceedings of the International Congress of Mathematicians, 15–22 August*, pages 23–35, Institut Mittag-Leffler, Djursholm, Sweden, 1962, Stockholm 1963.

[6] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:2000, 2000.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings, IBM Workshop on Logics of Programs, May 1981*, pages 52–71, 1982.

[8] E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *Proc. of CAV*, pages 263–277, 2009.

[9] B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Proc. of LICS*, pages 321–330, 2005.

[10] B. Finkbeiner and S. Schewe. SMT-based synthesis of distributed systems. In *Proc. of AFM*, pages 69–76. ACM Press, 2007.

[11] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[12] O. Kupferman and M. Y. Vardi. Synthesis with incomplete informatio. In *Proceedings of the 2nd International Conference on Temporal Logic (ICTL 1997), 14–18 July, Manchester, UK*, pages 91–106, 1997.

[13] O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001)*, pages 389–398, 2001.

[14] P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001)*, pages 396–407, 2001.

[15] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68–93, Jan. 1984.

[16] G. L. Peterson and J. H. Reif. Multiple-person alternation. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS 1979)*, pages 348–363, 1979.

[17] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57, 1977.

[18] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL 1989)*, pages 179–190, 1989.

[19] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proceeding of the 16th International Colloquium on Automata, Languages and Programming (ICALP 1989)*, pages 652–671, 1989.

[20] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS 1990)*, pages 746–757, 1990.

[21] M. O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the American Mathematical Society*, 141:1–35, 1969.

[22] M. O. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *Regional Conference Series in Mathematics*. 1972.

[23] J. H. Reif. The complexity of two-player games of incomplete information. *Journal of Computer and System Sciences*, 29(2):274–301, 1984.

[24] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Sceince, Rehovot, Israel, 1992.

[25] S. Schewe and B. Finkbeiner. Synthesis of asynchronous systems. In *Proc. of LOPSTR*, pages 127–142, 2006.

[26] S. Schewe and B. Finkbeiner. Bounded synthesis. In *Proc. of ATVA*, pages 474–488, 2007.

[27] I. Walukiewicz and S. Mohalik. Distributed games. In *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2003)*, pages 338–351, 2003.