

Property Synthesis

Barbara Jobstmann

Barbara Jobstmann
Graz University of Technology
bjobst@ist.tugraz.at
Estimated graduation date: December 2006
Advisor: Roderick Bloem
ASP-DAC PhD Forum: No
DATE PhD Forum: No

1 Motivation

Imagine you need to develop a new chip and all you have to do is to write its specification. Synthesizing hardware from its specification will take the design process to the next level of abstraction.

On the block level, this vision is now getting closer to reality. Given a formal specification written in a temporal logic, property synthesis constructs a design on the RT level that adheres to the specification. The formal specification is given as a set of properties that are defined over input and output signals. The specification language is a linear logic such as LTL, PSL, or SVA. The generated design is described in an HDL (e.g., Verilog).

Property synthesis provides the user with a functionally correct design right after the specification phase. The obvious advantage is that the generated system is guaranteed to be correct by construction. Thus, for blocks that have only functional requirements, hand-coding and design verification are obsolete. Blocks that are subject to non-functional requirements for instance on timing, space, or power usage need further optimizations. However, for those blocks a functional correct prototype is also useful. It allows for integration testing even if some blocks are not optimized completely. This helps to catch high-level design errors in a very early stage of the design phase.

Another benefit is that property synthesis helps to validate the specification. Simulating the functional prototype provides a simple way to check whether the specification fulfills the design intent. Besides, the user gets an immediate feedback on the specification she has written. Furthermore, some properties in the specification can be unrealizable, which means that those properties are violated in all possible designs. Those properties are useless in a formal specification and should be left out. Property synthesis answers the question of realizability and can identify unrealizable parts of the specification.

Property synthesis concentrates on specifications written in LTL, PSL, or SVA. These languages are well-known, easy to write and understand, and still expressible enough. LTL synthesis is known to have doubly exponential worst case complexity. The doubly exponential completeness and the very complex algorithms (such as Safra's determinization construction) used in the standard approach to LTL synthesis discouraged many researcher from pursuing the topic for practical use. Nevertheless, we found a way, described in Section 2, to handle the complexity.

There is plenty of theoretical work on LTL synthesis and a few implementations covering subsets of LTL but to our knowledge no implementation for the complete language. Recent work of Amir Pnueli handles the most general subset. His approach is only applicable to specifications expressible with generalized Streett[1] acceptance condition. Besides, suitable specifications have to be rewritten to a particular syntax in order to be synthesized.

My aim is to make property synthesis more practical by providing an implementation of a complete LTL synthesis approach. The implementation will provide a definite answer to the realizability question, a correct design if the specification is realizable, and debug information on the specification in the other case. Despite the high complexity, the implementation has to keep the intermediate data structures and the final design manageable. That means the doubly exponential blow-up will only occur if necessary. Since the problem is doubly exponential complete, there are properties that require the blow-up, i.e., the size of any design fulfilling those properties will be at least doubly exponential in the length of the specification. This implies that hand-design cannot do better. In addition, the complexity of LTL synthesis is given with respect to the specification and not with respect to the size of the generated design. In verification the limiting factor is always the size of the design and not the specification. So, if the size of the generated design is as small as possible, synthesis is not harder than verification.

2 Realizing Property Synthesis

I have started my work on synthesis by developing an efficient algorithm for a subset of LTL. More precisely, the set of specifications expressible by deterministic Büchi word

automata. Together with Griesmayer, Bloem, and Staber, I applied the approach to the area of program repair [1] and extended it to fault localization by combining finding and fixing faults in [2, 3]. Our key idea was to state fixing faults as a controller synthesis problem and use synthesis techniques. After that I started to work on a complete realization.

Recently, Kupferman and Vardi [4] have described a new approach to synthesis that goes through universal co-Büchi and weak alternating tree automata. In contrast to previous approaches it avoids Safra's determinization construction and parity games. My current work is based on this approach. I am developing optimization and approximation algorithms for the various steps of the approach to keep the (intermediate) results tractable and make an efficient implementation possible.

The approach uses the following steps to check whether a specification φ is realizable [4, 5]: (1) construct a nondeterministic Büchi word automaton B for $\neg\varphi$, (2) use B and a partition of the signals into input and output signals to construct a universal co-Büchi tree automaton C that accepts all realizations of φ , (3) translate C to an alternating weak tree automaton W , (4) translate W into a nondeterministic Büchi tree automaton T , and (5) check that the language of T is nonempty.

Beside the use of automata minimization algorithms such as direct and fair simulation minimizations in Step 1, we propose the following optimizations. Based on our algorithm for the program repair application, which uses nondeterministic Büchi games, we can reduce the state space of the automaton in Step 2. We extend the notion of direct and fair simulation to tree automata and use it and other techniques to minimize the size of the automaton in Step 3. For Step 4 and 5, we propose a combined iterative algorithm, which calculates the Büchi game of Step 5 during the construction of the nondeterministic Büchi tree automaton T . We expect it to perform better than an algorithm that processes the two steps sequentially. Finally, the output of Step 5 is either the fact that the specification is unrealizable or a witness for the language. The witness is a tree and corresponds to a strategy for the system that determines how to fulfill the specification depending on the input signals seen so far. This strategy is subject to further automata minimizations.

My current implementation includes the main approach and about half of the proposed optimizations. The preliminary experimental results are very promising. As expected, it turns out that the doubly exponential blow-up is avoided in many cases.

In my future work I will focus on reasons for unrealizability, because it is very hard to identify them just by looking at the properties. In the following, I summarize the reasons we have identified so far and propose appropriate debug information for it.

The simplest and easiest-to-proof reason for unrealizability is contradiction between several properties or subformulas. In this case, the user is presented with the affected formulas. Another possibility is the existence of a single input sequence that cannot be extended with a suitable output sequence to fulfill the specification. Providing the user with this input sequence and the violated properties helps the user to identify the fault in the specification. Even if a specification is satisfiable and each input sequence has a suitable output sequence, it may still be unrealizable. In this case, there exists no strategy for the system, that mean we cannot determine values for the outputs that fit all future input sequences based on the inputs seen so far. Only a system that could adjust its decisions to future input values would be able to realize such a specification. It is difficult to present the user with an example of this kind of unrealizability, because here the witness is a tree, instead of a single trace as in the previous case. This tree corresponds to a strategy for the environment that tells how to force a violation of the specification. We use this strategy to construct a design for the environment. The user can simulate this design and can try to find values for the output signals to fulfill the specification and she will see that this task is impossible. The proposed debug information will be extracted from the current implementation.

Property synthesis is expected to be very valuable to the hardware design and verification process. My work brings property synthesis one step closer to practical use.

References

- [1] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *17th Conference on Computer Aided Verification (CAV '05)*, pages 226–238, 2005.
- [2] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *13th Conference on Correct Hardware Design and Verification Methods (CHARME '05)*, pages 35–49, 2005.
- [3] S. Staber, B. Jobstmann, and R. Bloem. Diagnosis is repair. In *16th International Workshop on Principles of Diagnosis*, pages 169–174, 2005.
- [4] O. Kupferman and M. Vardi. Safraless decision procedures. In *Symposium on Foundations of Computer Science (FOCS'05)*, pages 531–542, 2005.
- [5] R. Bloem, B. Jobstmann, and A. Pnueli. Property-based logic synthesis for rapid design prototyping. Technical Report Prosyd D2.2.1, Prosyd, Graz University of Technology, 2005. <http://www.prosyd.org/twiki/pub/Public/DeliverablePageWP2/Prosyd2.2.1.pdf>.