

Research Summary

Game Theory for Reliable Computer Systems

Barbara Jobstmann

January 5, 2009

My recent research develops game theory techniques and applies them to the problem of constructing reliable computer systems. Game theory is gaining importance in many disciplines, including economics, sociology, political and sciences. In my research, I have used games to automatically construct, correct, and analyze computer systems and their specifications.

The games I study are turn-based games. Each game is played on a finite automaton; each state in the automaton is owned by one of the players. Initially, a token is put on a state. The player owning the state with the token can move the token along any of the outgoing edges to a next state, and the next turn starts. In general, plays are infinite sequences of turns. The number of players and their objectives vary with the application. The fundamental techniques used in this branch of game theory go back to Church [Chu62], McNaughton [McN66], Büchi and Landweber [BL69], and Rabin [Rab69]. These results have shown that game theory is in principle applicable to problems such as synthesis of systems from specification. However, previous techniques failed to yield practical algorithms.

With my collaborators, I made the following contributions to the area of game theory and its applications to computer systems:

1. We were the first to develop a tool [JSGB09] that is able to construct circuits from their specifications [BGJ⁺07b]. We developed optimizations to construct and solve these types of games more efficiently [JB06a]. This enabled us to design and implement the first tool to **synthesize finite-state systems from a specification** given by an arbitrary formula in Linear Temporal Logic [Pnu77].
2. We were the first to show the applicability of game theory to **fault localization** [SJB05b, SJB05a] and automatic **repair of programs** [JGB05, JSGB09]. We developed a new approach to deal with faulty systems and implemented it in the model checker VIS[B⁺95].
3. We showed how a game approach can be used to improve the **quality of specifications** in two ways:
 - We introduced a new, more appropriate, notion of equivalence between specifications of systems [GBJV08]. We provide lower and upper bounds and an optimal algorithm for the corresponding decision problems of our new notion.
 - We developed an approach that, given a specification of a system, constructs necessary assumptions on the environment of the system, automatically turning formally useless specifications into useful ones [CHJ08].
4. We contributed to the area of **component-based design** by extending an existing interface theory with a “shared refinement” operator [DHJP08]. The notion of refinement is stated as a game. Our new operator allows us to model component reuse, which was not possible in previous theories.

I did part of this work [JGB05, SJB05b, SJB05a, JB06a, JB06b, Job06, BGJ⁺07b, BGJ⁺07a, Job07, JGWB07] during my Ph.D. studies at the Graz University of Technology in Graz, Austria. I obtained further results [GHJS08, GBJV08, CHJ08, DHJP08, JSGB09] as a postdoctoral researcher at École Polytechnique Fédérale de Lausanne (EPFL), Switzerland. I next describe our selected contributions in more detail.

Synthesis for Automatic Construction of Systems

I made contributions to the synthesis problem, whose goal is to automatically construct the program from given specification. In contrast to verification approaches such as model checking [CE81, QS81] where the developer must provide both the specification and the program, the synthesis approach aims to relieve the developer of the burden of developing the program.

The problem of deriving a (non-terminating) reactive system from formulas in temporal logics has been first stated by Church [Chu62] in the early 60s. Subsequently, Büchi and Landweber [BL69] and Rabin [Rab69] presented solutions to this problem. Pnueli and Rosner [PR89] provided lower bounds and an optimal algorithm for realizing specification in Linear Temporal Logic (LTL) [Pnu77]. The key observation to LTL synthesis is that a reactive system with a set of input signals \mathcal{I} and set of output signals \mathcal{O} corresponding to a labeled infinite regular tree, where each node has $2^{\mathcal{I}}$ successors and is labeled with a letter in $2^{\mathcal{O}}$. Intuitively, every path of the tree corresponds to a sequence of input values the system has received and the labels on the nodes are the output values it produces. Given an LTL formula, we can construct a tree automaton that accepted exactly those trees that correspond to systems that implement the specification [PR89]. Thus, the synthesis problem can be reduced to the emptiness problem of tree automata, which in turn can be reduced to winning a turn-based game with two players [GH82]. Traditionally, this game is built by a chain of automata constructions involving an intricate determinization construction [Saf88] that is hard to implement and optimize. This construction together with a double exponential lower bound [Ros92] led to the view that LTL synthesis is only of theoretical interest.

Our research used two approaches to make LTL synthesis practical: optimizations of the general problem and algorithms for synthesis from an LTL subset.

Optimizations of the General Problem. We developed and implemented an approach to LTL synthesis [JB06a, JB06b] based on a recent algorithm [KV05] that uses a translation through universal co-Büchi tree automata and alternating weak tree automata. Our approach includes several optimization techniques for alternating tree automata, including a game-based approximation to language emptiness and a simulation-based optimization. Furthermore, we use an incremental algorithm to compute the emptiness of nondeterministic Büchi tree automata. All our optimizations are computed in time polynomial in the size of the automaton on which they are computed. We have applied our implementation to several examples and showed a significant improvement over the straightforward implementation. This work constitutes the first implementation of a synthesis algorithm for full LTL.

Synthesis from an LTL Subset. To increase the scalability of synthesis and make it applicable to construction of systems, we concentrated on useful LTL subsets to avoid the inherent complexity of general LTL synthesis. Researchers have developed several interesting classes of specifications with polynomial-time synthesis algorithms. A major progress has been made in [PPS06], where the authors propose to use formulas in Generalized Reactivity(1) (GR1) form and present an algorithm that is cubic in the state space of the design. We designed and implemented a tool based on [PPS06] and developed a method to construct actual circuits from the specification [BGJ⁺07b]. We demonstrate the application of our tool Anzu [JGWB07] using two examples from practice: 1) the AMBA AHB bus arbiter [ARM99] and 2) a generalized buffer carefully specified by IBM. The results we obtained using our tool are the first time that industrial though modest-size examples have been automatically synthesized from LTL specifications.

Finding and Repairing Faults

Our work on finding and repairing faults showed how to apply synthesis techniques in scenarios where the developer supplies a partial specification and a (potentially incorrect) version of the system. By eliminating the manual effort previously needed to fix and correct the program, our work goes further than existing bug finding techniques such as model checking.

Model checking has shown to be a very successful technique for verifying whether a given program adheres to the provided specification. The specification is a set of properties that the program should fulfill. It includes both simple properties such as assertions or invariants that specify desired relationships between the variables of the program, but also more general properties,

such as the correct interaction between the program and its environment. The specifications are often expressed in Linear Temporal Logic (LTL) [Pnu77].

Our research addresses two issues left unsolved by the model checking techniques: identifying parts of the program causing the bug, and identifying the repair for the bug. In [JGB05], we present an approach to repair a program that does not satisfy its specification. We view each program as consisting of components. The choice of components depends on the level of abstraction of the program: on the source code level the components are expressions; on the gate level the components are gates.

Our first, simpler, approach assumes as input a set of components that are suspected to cause the fault. The key idea is to replace these components by completely unconstrained components and synthesize the components so that the entire program satisfies the specification. To synthesize the components, we view the modified program as a game. We let Player 1 choose how to restrict the new components. Player 2 can choose the input values of the program. The winning objective of Player 1 is to satisfy the given specification independent of the values Player 2 chooses. If Player 1 can win this game, we have found a repair for the components. We show how to use a winning strategy of Player 1 to extract a repair and map it back to the program.

Our more sophisticated approach [SJB05b, SJB05a] does fault localization and lifts the restriction of having an initial set of suspected faulty components. In this work, we say that a component may be responsible for a fault if and only if it can be replaced by an alternative that makes the system correct. This allows us to introduce additional choice for Player 1. In the extended game, Player 1 can first choose the component to be replaced, and as the game proceeds Player 1 determines the replacement. In [JSGB09], we summarize our approach and show its applicability to several examples including the TCAS (Traffic Collision Avoidance System) task of the Siemens test suite [DER05].

Analysis of Specifications

We have recently developed two approaches to analyze specifications and improve their quality. Given that the quality of a specification has a huge influence on the verification, synthesis, and repair process, we believe our techniques to have a significant potential.

Our first approach [GBJV08] shows that the usual trace-based notions of implication and equivalence for LTL are, in fact, too strong, and should be replaced by the weaker notions. We introduce appropriate weaker notions called open implication and open equivalence. Open implication is more difficult to compute than standard implication, but has advantages in both model checking and synthesis. We study the difference between trace-based equivalence and open equivalence and describe an algorithm to compute open implication of LTL with asymptotically optimal complexity. We also show how to compute open implication while avoiding Safra's construction. We have implemented an open-implication solver for GR1 specifications [PPS06]. Using the AMBA AHB bus arbiter [ARM99] case study, we show that open equivalence justifies the use of an alternative specification and allows us to automatically synthesize orders of magnitude more compact implementations in orders of magnitude less time, enabling scaling to larger design.

Our second approach [CHJ08] shows how to automatically correct unrealizable specifications. Our observation is that many specifications written by users of synthesis tools are unrealizable, which means that no system can implement the specification. A common reason for unrealizability is that the assumptions on the environment in the specification of the system are incomplete, making the specification too strong. We address the problem of correcting an unrealizable specification φ by computing an environment assumption ψ such that the new specification $\varphi \rightarrow \psi$ is realizable. Our goal is to construct an assumption ψ that 1) constrains only the environment and not the system and 2) is as weak as possible. We present a two-step algorithm for computing assumptions. The algorithm operates on the same game graph that is used to answer the realizability question. First, the algorithm computes a safety assumption that removes a minimal set of environment edges from the graph. Second, the algorithm computes a liveness assumption that puts fairness conditions on some of the remaining environment edges. We show that the problem of finding a minimal set of fair edges is computationally hard, and we use probabilistic games to compute a locally minimal fairness assumption.

Component-Based Design

In the area of techniques for designing systems from components, I have contributed to enabling a correct use of components that should simultaneously satisfy multiple specifications, expressed as interfaces. Interface theories [dAH01] were shown useful for component-based design because they support incremental design and independent implementability. Incremental design refers to the ability to check in advance, without knowing the interfaces of all components, if two particular interfaces are compatible. This allows to decompose a design in a top-down fashion. Independent implementability means that compatible interfaces can be refined separately, while maintaining their compatibility. Therefore, it provides a way to implement each component independently of the remaining parts of the design. Interface theories define the notion of compatibility and refinement using game theory.

Our work on component-based design [DHJP08] shows how to deal with components that implement several interfaces, such as components reused in several parts of a design. We show that previously known interface theories provide no formal support for component reuse. We introduce a new operator, shared refinement, that supports such reuse. In addition to combining interface descriptions from different parts of the design, our shared refinement operator enables combining different aspects of a single component, such as functionality, timing, and power consumption, each of them given by an interface. We give both stateless and stateful examples for interface theories with component reuse. To illustrate component reuse in interface-based design, we showed how the stateful theory provides a natural framework for specifying PCI bus clients and refining such specifications.

References

- [ARM99] ARM Ltd. AMBA specification (rev. 2). Available from www.arm.com, 1999.
- [B⁺95] R. K. Brayton et al. VIS: A system for verification and synthesis. Technical Report UCB/ERL M95/104, Electronics Research Lab, Univ. of California, December 1995.
- [BGJ⁺07a] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *In Proceedings of the Design, Automation and Test in Europe*, pages 1188–1193, 2007.
- [BGJ⁺07b] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. In *6th International Workshop on Compiler Optimization Meets Compiler Verification*, 2007. Electronic Notes in Theoretical Computer Science <http://www.entcs.org/>.
- [BL69] J. R. Büchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.
- [CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings Workshop on Logics of Programs*, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131.
- [CHJ08] K. Chatterjee, T. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *International Conference on Concurrency Theory (CONCUR)*, pages 147–161, 2008.
- [Chu62] A. Church. Logic, arithmetic and automata. In *Proceedings International Mathematical Congress*, 1962.
- [dAH01] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *Proceedings of EMSOFT: Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2001.
- [DER05] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10:405–435, 2005.
- [DHJP08] L. Doyen, T. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *International Conference on Embedded Software (EMSOFT)*, pages 79–88, 2008.
- [GBJV08] K. Greimel, R. Bloem, B. Jobstmann, and M. Vardi. Open implication. In *Proc. Int. Colloquium on Automata, Languages and Programming (ICALP'08)*, pages 361–372, 2008. LNCS 5126.

- [GH82] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proc. 14th ACM Symp. Theory of Comp.*, pages 60–65, San Francisco, CA, 1982.
- [GHJS08] R. Guerraoui, T. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *Programming Language Design and Implementation (PLDI)*, pages 372–382, 2008.
- [JB06a] B. Jobstmann and R. Bloem. Game-based and simulation-based improvements for LTL synthesis. In *3rd Workshop on Games in Design and Verification (GDV'06)*, 2006.
- [JB06b] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 117–124, 2006.
- [JGB05] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In K. Etessami and S. K. Rajamani, editors, *17th Conference on Computer Aided Verification (CAV'05)*, pages 226–238. Springer-Verlag, 2005. LNCS 3576.
- [JGWB07] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification*, pages 258–262, 2007.
- [Job06] B. Jobstmann. Property synthesis. In *9th SIGDA Ph.D. Forum at the Design Automation Conference (DAC 2006)*, San Francisco, California, USA, 2006.
- [Job07] B. Jobstmann. *Applications and Optimizations for LTL Synthesis*. PhD thesis, Graz University of Technology, March 2007.
- [JSGB09] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *Journal of Computer and System Sciences*, 2009. Accepted for publication.
- [KV05] O. Kupferman and M. Y. Vardi. Safrless decision procedures. In *Foundations of Computer Science*, pages 531–542, Pittsburgh, PA, October 2005.
- [McN66] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.
- [PPS06] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380. Springer, 2006. LNCS 3855.
- [PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989.
- [QS81] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth Annual Symposium on Programming*, 1981.
- [Rab69] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.
- [Ros92] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.
- [Saf88] S. Safra. On the complexity of ω -automata. In *Symposium on Foundations of Computer Science*, pages 319–327, October 1988.
- [SJB05a] S. Staber, B. Jobstmann, and R. Bloem. Diagnosis is repair. In *16th International Workshop on Principles of Diagnosis*, pages 169–174, 2005.
- [SJB05b] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In D. Borrione and W. Paul, editors, *13th Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, pages 35–49. Springer-Verlag, 2005. LNCS 3725.
- [vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*. The MIT Press/Elsevier, Amsterdam, 1990.