Report from Dagstuhl Seminar 12152

Software Synthesis

Edited by

Rastislav Bodik[1], Sumit Gulwani[2], and Eran Yahav[3]

1 **University of California – Berkeley, US,** `bodik@cs.berkeley.edu`
2 **Microsoft Research – Redmond, US,** `sumitg@microsoft.com`
3 **Technion – Haifa, IL,** `yahave@cs.technion.ac.il`

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 12152 "Software Synthesis". During the seminar, several participants presented their current research, ongoing work and open problems were discussed. Abstracts of the presentations given during the seminar are put together in this paper.

The rise of multiprocesser computers and of software verification as applied in industry combine to create an opportune moment for software synthesis. To facilitate research in this area, research from several fields of Computer Science presented tutorials on techniques they developed. This lead to (1) the definition of what challenges synthesis has to tackle in the future and (2) insights into how the several fields of synthesis are related.

Finally, several groups described their experience with teaching synthesis to graduate and undergraduate students, demonstrating that synthesis is challenging for students but that they can also rise to the challenge and enjoy the field.

Seminar 09.– 13.04, 2012 – www.dagstuhl.de/12152
1998 ACM Subject Classification I.2.2 Program Synthesis
Keywords and phrases Software Synthesis, Verification and Model Checking, Theorem Proving, Program Analysis, Programming by Demonstration, Program Derivation, Compiler Optimization
Digital Object Identifier 10.4230/DagRep.2.4.21
Edited in cooperation with Christian von Essen

1 Executive Summary

Rastislav Bodik
Sumit Gulwani
Eran Yahav

Software verification and synthesis are founded on similar principles, yet verification has become industrial reality while successes of synthesis remain confined to a handful of domains. Still, recent years witnessed increased interest in software synthesis—a trend spurred by growing software complexity and simultaneously enabled by advances in verification, decision procedures, and machine learning. The goal of the seminar is to help the revival of software synthesis through intellectual exchange among experts in deductive synthesis, controller synthesis and the diverse spectrum of new synthesis efforts in inductive synthesis, auto-tuning, programming by demonstration and partial programming.

This is an opportune moment for software synthesis. First, multi-core processors are likely to make software development harder, motivating automatic construction of synchronization and communication code. Second, software verification and checking reached industrial maturity through judicious use of linguistic support, decision procedures, and dynamic analyses, inspiring solutions to open synthesis problems. Third, by incorporating verification into synthesis, we may be able to synthesize programs that are easier to verify than hand-written programs. Fourth, parallel computers enable search powerful enough for synthesis of well-tuned programs, as demonstrated by auto-tuners and super-optimizers. Finally, recent systems built on programming by demonstration make us hope that specification will be easier to write.

The seminar organizers hope to achieve the following goals:

- Offer brief tutorials on techniques developed by communities participating in the seminar.
- Develop a set of challenge problems for practical synthesis, a collection of practical problems solvable by (semi-)automatic synthesis in five years.
- Deepen the understanding of the relationships between the various approaches to synthesis. In particular, to what extent are the techniques developed by the respective communities independent from their driving applications? Understand strengths of the alternative approaches.
- Understand relationships and applicability of verification technology to software synthesis.
- Outline a syllabus for a graduate course in software synthesis.

2 Table of Contents

3 Overview of Talks

3.1 Re-Envisioning Lightweight Modeling

Krzysztof Czarnecki (University of Waterloo, CA)

Joint work of Czarnecki, Krzysztof; Antkiewicz, Michal; Bak, Kacper; Wasowski, AndrzeJ
Main reference K. Bak, K. Czarnecki, A. Wasowski, "Feature and meta-models in Clafer: mixed, specialized, and coupled," in Proc. of the Third Int'l Conf. on Software Language Engineering (SLE'10), Springer-Verlag, Berlin, Heidelberg, pp. 102–122, 2010.
URL http://dx.doi.org/10.1007/978-3-642-19440-5_7

Model-driven engineering (MDE) has failed to reach the masses. MDE is the use of models—reduced, purposeful, and comprehensible representations of a system and its environment—to describe, analyze and construct a system in a tool-supported way. It is a compelling idea designed to deal with the ever-growing complexity of today and tomorrow's software-intensive systems. Yet MDE remains confined to niches that represent only a small portion of the software industry today.

In the first part of the talk, I will attempt to uncover the causes for this failure. Two main forms of MDE exist today: using domain-specific modeling languages (DSMLs) or the UML, a general-purpose modeling language. Both forms have failed to stimulate a wide adoption of MDE. Domain-specific approaches are successful in specific areas only—thus, being confined to niches by definition. Another challenge to DSMLs is the high up-front investment necessary to develop them. General-purpose languages, such as UML, again by definition, are widely applicable but offer inevitably much less value in any particular context. To make the situation worse, UML and its associated tools are heavyweight, rendering UML-based MDE cost-ineffective in most cases.

In the second part of the talk, I will turn to lightweight modeling—an emerging form of modeling that may overcome the limitations of DSMLs and the UML. Lightweight modeling—defined by Zave as the use of small, abstract models and push-button verification—focuses on easy-to-use languages, models, and tools and thus improves the cost-effectiveness of general-purpose modeling. While lightweight modeling has had some successes in specific areas, such as network protocol design and verification, I believe that several improvements are necessary to make the technology attractive to a much wider audience. I will attempt to re-envision lightweight modeling from first principles, starting with a lightweight method and moving to language and tool design to support such a method. The suggested improvements will include support for transitioning from informal to formal, from concrete to abstract, and from general-purpose to domain-specific. They also include support for modeling and analyzing rich state and behavior and for co-evolution of abstractions and examples. I will illustrate some of these points using Clafer (see clafer.org), a lightweight modeling technology under development.

3.2 Repair of Sequential Programs

Jyotirmoy Deshmukh (University of Pennsylvania, US)

Joint work of Deshmukh, Jyotirmoy; Samanta, Roopsha; Emerson, E. Allen
Main reference R. Samnta, J. V. Deshmukh, E. Allen Emerson, "Automatic Generation of Local Repairs for Boolean Programs," in Proc. of the 2008 Int'l Conf. on Formal Methods in Computer-Aided Design, pp. 1–10, 2008.
URL http://dx.doi.org/10.1109/FMCAD.2008.ECP.31

Automatic techniques for software verification focus on obtaining witnesses of program failure. Such counterexamples often fail to localize the precise cause of an error and usually do not suggest a repair strategy. We present an efficient algorithm to automatically generate a repair for an incorrect sequential program with variables ranging over finite domains, where program correctness is specified using pre-conditions and post-conditions. Our approach draws on standard techniques from predicate calculus to obtain local annotations for the program statements, which are then used to generate a repair existence query for each program statement, which if successful, yields a repair. While our original work focused on Boolean programs, it is possible to extend this to programs with more general kind of variable types, as long as these correspond to finite domains.

3.3 Partial-Observation Stochastic Games: How to Win when Belief Fails

Laurent Doyen (ENS – Cachan, FR)

Joint work of Chatterjee, Krishnendu; Doyen, Laurent
Main reference K. Chatterjee, L. Doyen, "Partial-Observation Stochastic Games: How to Win when Belief Fails," in Proc. of LICS 2012. IEEE.
URL http://arxiv.org/abs/1107.2141

We consider two-player stochastic games played on finite graphs with reachability (and Buechi) objectives, and almost-sure winning (i.e., with probability 1), or positively winning (i.e., with positive probability).

We classify such games according to which player(s) have partial observation and whether the players use pure or randomized strategies.

Our main results for pure strategies and both positive and almost-sure winning are as follows: (1) When player 2 has perfect observation we show that belief-based strategies are not sufficient for player 1. We present an exponential upper bound on the memory needed by winning strategies, and we show that the problem of deciding whether player 1 wins is EXPTIME-complete. (2) When player 1 has perfect observation we show that non-elementary memory is both necessary and sufficient for winning strategies of player 1. (3) When both players have partial observation, we show that finite-memory strategies are sufficient.

We establish the equivalence of the almost-sure winning problems for pure strategies and for randomized strategies with actions invisible, which exhibits serious flaws in previous results in the literature: we show a non-elementary memory lower bound for almost-sure winning whereas an exponential upper bound was previously claimed.

3.4 Constraint Programming and Synthesis

Pierre Flener (Uppsala University, SE)

After a brief tutorial on the constraint programming (CP) paradigm for the modelling and solving of combinatorial problems, we argue that CP constitutes a form of program synthesis. Furthermore, we identify opportunities for applying synthesis techniques within the area of CP, as well as opportunities for applying CP within the area of synthesis.

3.5 Lessons Learned in the 1990s about Synthesis from Partial Specifications

Pierre Flener (Uppsala University, SE)

Joint work of Flener, Pierre; Schmid, Ute
Main reference P. Flener, U. Schmid, "An introduction to inductive programming," Artificial Intelligence Review, 29(1):45–62, March 2008.
URL http://dx.doi.org/10.1007/s10462-009-9108-7

I discuss the recommendations on program synthesis from partial specifications that I would have made in the late 1990s when I stopped active research in that area (because I moved to complete forms of specifications). These recommendations are drawn from various survey papers I wrote on the topic.

3.6 Course on Program Synthesis

Pierre Flener (Uppsala University, SE)

Main reference P. Flener, "Logic Program Synthesis from Incomplete Information," International Series in Engineering and Computer Science, Vol. 295, Springer-Verlag, 195
URL http://www.springer.com/computer/swe/book/978-0-7923-9532-4

From 1993 to 1997 I taught a course on program synthesis to graduate students at Bilkent University, a private elite university in Ankara, Turkey. The first half of the course was made of lectures on the background of deductive and inductive synthesis, mostly drawn from my 1995 book at Kluwer. The second half of the course was made of student presentations, each on a synthesiser the student read papers about, if not experimented with.

3.7 Spiral: Library Generation Through Autotuning, Rewriting, and Constraint Solving

Franz Franchetti (Carnegie Mellon University – Pittsburgh, US)

Main reference M. Půschel, F. Franchetti, Y. Voronenko, "Spiral," Encyclopedia of Parallel Computing, D. A. Padua (Editor).
URL http://users.ece.cmu.edu/~franzf/papers/spiral-enc11.pdf

Automatically achieving performance on par with human programmers on current and emerging parallel platforms is a key challenge for the automatic performance tuning community. With the Spiral system (www.spiral.net) we have shown that it is indeed possible to achieve performance portability across a wide range of parallel platforms from embedded processors to supercomputers at or above the performance level achieved by human experts (a.k.a. Black Belt Programmers), for a restricted set of algorithms. We will discuss our experience with building the Spiral system and adapting it to the ever changing landscape of parallel platforms.

3.8 On Optimal and Reasonable Control in the Presence of Adversaries

Oded Maler (VERIMAG – Gières, FR)

Main reference O. Maler, "On Optimal and Reasonable Control in the Presence of Adversaries," IFAC Annual Reviews in Control, 2007.
URL http://www-verimag.imag.fr/~maler/Papers/annual.pdf

This paper constitutes a sketch of a unified framework for posing and solving problems of optimal control in the presence of uncontrolled disturbances. After laying down the general framework we look closely at a concrete instance where the controller is a scheduler and the disturbances are related to uncertainties in task durations.

3.9 Automating End-User Programming for Smartphones

Vu Minh Le (University of California – Davis, US)

Joint work of Le, Vu; Gulwani, Sumit; Su, Zhendong

Program synthesis has shown recent promise in a few domain-specific settings. This paper identifies an emerging, important application domain for synthesis, viz., end-user programming for smartphones. We propose a new end-user programming model inspired by the search engine metaphor: (1) the end user programs by entering a set of keywords, which are mapped to a set of programming components, and (2) the synthesis system enumerates and ranks all valid compositions of these components in our domain specific language (DSL). The design of our DSL is based on an extensive study of mobile applications from various online forums. Our algorithm exploits the components' type information and the structural constraints imposed by our DSL to reduce the search space, and learns the user's intent by using a general ranking scheme. The system works extremely effectively in practice – over all of our collected benchmark examples, it is able to rank, in real time, the desired application among the top two choices. It is also easy to use – users can quickly and accurately select the relevant components and decide whether a solution meets their intent. We believe that our methodology is general enough to be applied to other end-user programming domains as well.

3.10 Synthesizing Algorithms to Solve Scheduling Problems from High-Level Models

Jean-Noël Monette (Uppsala University, SE)

Main reference J.-N. Monette, "Solving scheduling problems from high-level models," 4OR 9(3), pp. 317–320 (2011).
URL http://dx.doi.org/10.1007/s10288-010-0143-7

In this talk, I present a synthesis approach for a specific domain, namely (offline) scheduling. Scheduling problems are described using a declarative modelling language. The structure of the model is then analysed to classify the problem into some predefined classes of problems. The synthesiser has a strategy associated with each class of problem. A strategy a mainly the skeleton of an algorithm, that can be completed by more information on the structure of the problem and with data particular to the given instance. We developed a prototype, Aeon, that synthesis algorithms based on constraint programming, constraint-based local search, and greedy approaches. Experimental results show that this approach is competitive with the state-of-the-art on part of the benchmark problems.

3.11 Complete Functional Synthesis for Linear Integer Arithmetic

Ruzica Piskac (MPI für Softwaresysteme – Saarbrücken, DE)

Joint work of Kuncak, Viktor; Mayer, Mikael, Piskac, Ruzica; Suter, Philippe
Main reference V. Kuncak, M. Mayer, R. Piskac, P. Suter, "Complete functional synthesis," PLDI 2010: 316–329.
URL http://dx.doi.org/10.1145/1809028.1806632

In this talk we describe Comfusy, an extension to the Scala compiler and a tool for complete functional synthesis. Comfusy accepts as an input expressions with input and output variables specifying relations on integers and sets. Comfusy outputs code that computes output values as a function of input values. In addition, it also outputs the preconditions that the input variables have to satisfy in order for a solution to exist. In this talk we describe the algorithm that we use for complete functional synthesis of linear integer arithmetic. The algorithm relies on the effective handling of equalities, based on an extension of the Extended Euclidean Algorithm. The inequalities are processed in a similar style as in Fourier-Motzkin eliminations.

3.12 Interactive Synthesis of Code Snippets (Tool Demo)

Ruzica Piskac (MPI für Softwaresysteme – Saarbrücken, DE)

Joint work of Gvero, Tihomir; Kuncak, Viktor; Piskac, Ruzica
Main reference T. Gvero, V. Kuncak, R. Piskac, "Interactive Synthesis of Code Snippets," CAV 2011: 418–423.
URL http://dx.doi.org/10.1007/978-3-642-22110-1_33/

We present a tool that synthesizes and suggests valid expressions of a given type. Our tool supports polymorphic type declarations and can synthesize expressions containing methods with any number of arguments and any depth. Our synthesis approach is based on a quantitative generalization of the type inhabitation problem with weighted type assignments. The algorithm that we use is inspired by first-order resolution.

3.13 Synthesis Course at the Spring Academy of the German National Academic Foundation

Ruzica Piskac (MPI für Softwaresysteme – Saarbrücken, DE)

In this talk we give a report on the synthesis course given at the Spring Academy of the German National Academic Foundation in March 2012 in Papenburg. We describe the background of the participating students and give an outline of the course. The course topics came from reactive synthesis as well as from software synthesis. We also give outlines of the projects that the students did in the course.

3.14 Synthesizing Software Verifiers from Proof Rules

Corneliu Popeea (TU München, DE)

Automatically generated tools can significantly improve programmer productivity. For example, parsers and dataflow analyzers can be automatically generated from declarative specifications in the form of grammars, which tremendously simplifies the task of implementing a compiler. In this paper, we present a method for the automatic synthesis of software verification tools. Our synthesis procedure takes as input a description of the employed proof rule, e.g., program safety checking via inductive invariants, and produces a tool that automatically discovers the auxiliary assertions required by the proof rule, e.g., inductive loop invariants and procedure summaries. We rely on a (standard) representation of proof rules using recursive equations over the auxiliary assertions. The discovery of auxiliary assertions, i.e., solving the equations, is based on an iterative process that extrapolates solutions obtained for finitary unrollings of equations. We show how our method synthesizes automatic safety and liveness verifiers for programs with procedures, multi-threaded programs, and functional programs. Our experimental comparison of the resulting verifiers with existing state-of-the-art verification tools confirms the practicality of the approach.

3.15 Towards Algorithmic Synthesis of Synchronization for Shared-Memory Concurrent Programs

Roopsha Samanta (Univ. of Texas at Austin, US)

Main reference R. Samanta, "Towards Algorithmic Synthesis of Synchronization for Shared-Memory Concurrent Programs," in Proc. of Workshop on Synthesis (SYNT), Jul 7–8, 2012, Berkeley, CA.

We present a framework that takes a concurrent program composed of unsynchronized processes, along with a temporal specification of their global concurrent behaviour, and automatically generates a concurrent program with synchronization ensuring correct global behaviour. Our methodology supports finite-state concurrent programs composed of processes that may have local and shared data variables, may be straight-line or branching programs, may be ongoing or terminating, and may have program-initialized or user-initialized variables. The specification language is an extension of propositional Computation Tree Logic (CTL) that enables easy specification of safety and liveness properties over control and data variables. The framework also supports synthesis of synchronization at different levels of abstraction and granularity.

3.16 Synthesis of Succinct Systems

Sven Schewe (University of Liverpool, GB)

Joint work of Fearnley, John; Peled, Doron; Schewe, Sven
Main reference J. Fearnley, D. Peled, S. Schewe, "Synthesis of Succinct Systems," CoRR abs/1202.5449: (2012)
URL http://arxiv.org/abs/1202.5449

Synthesis of correct by design systems from specification has recently attracted much attention. The theoretical results imply that this problem is highly intractable, e.g., synthesizing a system is 2EXPTIME-complete for an LTL specification, and EXPTIME-complete for a CTL specification. However, an argument against it is that the temporal specification is highly compact, and the complexity reflects the large size of the system constructed. In that respect, the complexity should, perhaps, be specified relative to the size of the minimal satisfying system. A careful observation reveals that the size of the system is presented in such arguments as the size of its state space. This view is a bit nonstandard, in the sense that the state space can be exponentially larger than the size of a reasonable implementation such as a circuit or a program. Although this alternative measure of the size of the synthesized system is more intuitive (e.g., this is the standard way model checking problems are measured), research on synthesis has so far stayed with measuring the system in terms of the explicit state space. This raises the question of whether or not there always exists a small system. In this paper, we show that this is the case if, and only if, PSPACE = EXPTIME.

3.17 Algorithmic Synthesis for Biology

Saurabh Srivastava (University of California – Berkeley, US)

Joint work of Srivastava, Saurabh; Bodik, Ras; Seshia, Sanjit; Anderson, Chris

In this talk I will present a case for the application of program synthesis to synthetic biology. We believe synthesis of chemical reaction pathways can be one of the most exciting applications of program synthesis as the resulting pathways may suggest previously unknown paths to life-saving drugs, green fuels, and green polymers.

I will present one formulation of the pathway search problem as the synthesis of a acyclic program with graph transformations as the core primitive operators. Each operator comes from a hierarchy of operators derived from naturally observed transforms. I will also describe aspects of modularity in the system, and contrast against the state-of-the-art to motivate how beneficial synthesis can be in this domain.

3.18 Data Repair using Program Synthesis

Saurabh Srivastava (University of California – Berkeley, US)

In this talk we present an experience report in the domain of data repair that motivates a very obvious synthesis problem. Consider the problem of writing an analysis over a very long stream of data. We can refactor the analysis by taking the precondition checks out of it; and using it as a preprocessing step. We can then use the refactored precondition check as the specification for synthesizing a repairer. For each precondition-failing data row we can generalize from the trace by taking its symbolic execution path condition. Then we learn from the example and the user provided fix for it, a generalized fix using standard example-based synthesizers. Aggregating these guarded fixes provides the repairer.

3.19 Combining Synthesis Procedures

Philippe Suter (EPFL – Lausanne, CH)

Joint work of Suter, Philippe; Kuncak, Viktor

Recent work proposed synthesis procedure for functional synthesis over unbounded domains, such as linear arithmetic over real numbers or integers. A synthesis procedure acts as a compiler for declarative specifications. It accepts a formula describing a relation between inputs and outputs, and generates a function implementing this relation. This presentation shows how to combine synthesis procedures in a sound and complete way, providing a counterpart of the Nelson-Oppen combination technique in the area of synthesis.

3.20 Kaplan – Constraints as Control

Philippe Suter (EPFL – Lausanne, CH)

Joint work of Köksal, Ali Sinan; Kuncak, Viktor; Suter, Philippe;
Main reference Constraints as Control, POPL 2012
URL http://dx.doi.org/10.1145/2103656.2103675

We present an extension of Scala that supports constraint programming over bounded and unbounded domains. The resulting language, Kaplan, provides the benefits of constraint programming while preserving the existing features of Scala. Kaplan integrates constraint and imperative programming by using constraints as an advanced control structure; the developers use the monadic 'for' construct to iterate over the solutions of constraints or branch on the existence of a solution. The constructs we introduce have simple semantics that can be understood as explicit enumeration of values, but are implemented more efficiently using symbolic reasoning.

3.21 Fair Synthesis for Distributed Asynchronous Systems

Nathalie Sznajder (UPMC – Paris, FR)

Joint work of Gastin, Paul; Sznajder, Nathalie
Main reference P. Gastin, N. Sznajder, "Fair Synthesis for Distributed Asynchronous Systems," Transactions on Computational Logic.
URL http://tocl.acm.org/accepted/TOCL-2010-0004.pdf

We study the synthesis problem in an asynchronous distributed setting: a finite set of processes interact locally with an uncontrollable environment and communicate with each other by sending signals – actions controlled by a sender process and that are immediately received by the target process. The fair synthesis problem is to come up with a local strategy for each process such that the resulting fair behaviors of the system meet a given specification. We consider external specifications. External means that specifications only relate input and output actions from and to the environment and not internal signals exchanged by processes. We also ask for some natural closure properties of the specification. We present this new setting for studying the fair synthesis problem for distributed systems, and give decidability results for the subclass of networks where communications happen through a strongly connected graph. We claim that this framework for distributed synthesis is natural, convenient and avoids most of the usual sources of undecidability for the synthesis problem. Hence, it may open the way to a decidable theory of distributed synthesis.

3.22 Code checking, angelic execution, debugging and synthesis with Kodkod

Emina Torlak (University of California – Berkeley, US)

Main reference E. Torlak, "A constraint solver for software engineering: finding models and cores of large relational specifications," PhD Thesis. Massachusetts Institute of Technology, 2009.
URL http://dl.acm.org/citation.cfm?id=1713778

In this talk, I will present a brief tutorial on using Kodkod for code checking, angelic execution, debugging, and synthesis. Kodkod is an efficient SAT-based constraint solver for first order logic with relations, transitive closure, and partial models. It provides analyses for both satisfiable and unsatisfiable problems: a finite model finder for the former and a minimal unsatisfiable core extractor for the latter. Kodkod has been used in many applications, including code checking, declarative programming, test-case generation, and lightweight analysis of formal specifications written in Alloy, Isabelle/HOL, and UML.

3.23 Some synthesis problems from the embedded systems domain

Stavros Tripakis (University of California – Berkeley, US)

Joint work of Tripakis, S.; Bui, D.; Geilen, M.; Henzinger, T. A.; Lee, E. A.; Lickly, B.; Rodiers, B.; Andrade, H.; Ghosal, A.; Limaye, R.; Ravindran, K.; Wang, G.; Yang, G.; Kornerup, J.; Wong, I.; Puri, A.; Varaiya, P.; Lublinerman, R.; Szegedy, C.; Wiggers, M.

We briefly describe some synthesis problems that we have come across while working in the general area of embedded systems.

Interface Synthesis

Modern system design languages are *component-based*: large systems are designed by composing smaller (and simpler) subsystems, which in turn are composed by even smaller subsubsystems, and so on up to some basic components available as primitives from a library.

The concept of *interface* is generally used to capture the mechanisms by which a component interacts with its environment. But it has also been used recently in the broader context of *interface theories* [1], as a representation of relevant information about a component. An interface can therefore be seen as an abstraction of a component: the interface maintains only what is relevant (in particular, regarding interaction between the component and its environment) and hides the rest.

The goal of interface synthesis is to automatically produce the interface of a component given the interfaces of its subcomponents. Exactly what an interface is and how it can be synthesized, varies greatly depending not only on the syntax and semantics of the component language, but also on the purpose for which the interface is to be used. In a series of works we have studied the interface synthesis problem for various languages from the embedded systems domain: for synchronous formalisms such as Simulink we devised interfaces suitable for code generation [6, 5, 4] as well as for verification and refinement purposes [10]; for asynchronous dataflow formalisms such as SDF [3] we devised interfaces suitable for code generation [9] and others for performance analysis and refinement [2]. The above works include algorithmic techniques for interface synthesis. These can be viewed as automated abstraction methods. Although automated abstraction is a hard problem for software in general, it can be efficiently (and sometimes even optimally) solved in the context of the above domain-specific languages. It would be interesting to see how far this agenda can be extended, to cover other hierarchical and component-based formalisms.

Synthesizing the Alternating Bit Protocol

The Alternating Bit Protocol (ABP) is a basic mechanism for reliable transmission of data over an unreliable channel. Can such a protocol be devised automatically? The problem can be cast as a decentralized controller synthesis problem [7]. It would be interesting to see whether state-of-the-art synthesis techniques can address this challenge problem.

Glue Design

Component-based design is prevalent in the hardware design domain, where complex pieces of hardware are designed from more basic components, often described as “IP (intellectual property) blocks”. One of the difficult parts in this design process is to design the “glue” that combines the IP blocks. This glue usually consists in buffers and control logic that governs the execution and data exchanges between the IP blocks. Careful design of the glue is a must in order for the end-to-end system to be correct (are all data exchanged in an intact manner?) and also achieve performance requirements (is throughput sufficient? is buffer space minimized?).

Glue design is currently a mostly manual process. High-level models are sometimes used to make the process partly automatic, however, these models must typically be built manually, and they often result in suboptimal or even incorrect glue [8]. Can glue design by case as a tractable synthesis problem?

References

1 L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *EMSOFT'01*. Springer, LNCS 2211, 2001. 3.23
2 M. Geilen, S. Tripakis, and M. Wiggers. The earlier the better: A theory of timed actor interfaces. In *14th Intl. Conf. Hybrid Systems: Computation and Control (HSCC'11)*. ACM, 2011. 3.23
3 E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. 3.23
4 R. Lublinerman, C. Szegedy, and S. Tripakis. Modular Code Generation from Synchronous Block Diagrams – Modularity vs. Code Size. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 78–89. ACM, January 2009. 3.23
5 R. Lublinerman and S. Tripakis. Modular Code Generation from Triggered and Timed Block Diagrams. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 147–158. IEEE CS Press, April 2008. 3.23
6 R. Lublinerman and S. Tripakis. Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams. In *Design, Automation, and Test in Europe (DATE'08)*, pages 1504–1509. ACM, March 2008. 3.23
7 A. Puri, S. Tripakis, and P. Varaiya. Problems and Examples of Decentralized Observation and Control for Discrete Event Systems. In B. Caillaud, P. Darondeau, L. Lavagno, and X. Xie, editors, *Synthesis and Control of Discrete Event Systems*. Kluwer, 2001. 3.23
8 S. Tripakis, H. Andrade, A. Ghosal, R. Limaye, K. Ravindran, G. Wang, G. Yang, J. Kornerup, and I. Wong. Correct and non-defensive glue design using abstract mod-

els. In *9th Intl. Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 2011. 3.23

9 S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee. Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, Accepted for publication, Dec 2010. 3.23

10 S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A theory of synchronous relational interfaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(4), July 2011. 3.23

3.24 Abstraction-Guided Synthesis of Synchronization

Eran Yahav (Technion – Haifa, IL)

Joint work of Vechev, Martin; Yahav, Eran; Yorsh, Greta
Main reference M. Vechev, E. Yahav, G. Yorsh, "Abstraction-Guided Synthesis of Synchronization," POPL'10.
URL http://www.cs.technion.ac.il/~yahave/papers/popl10.pdf

We present a novel framework for automatic inference of efficient synchronization in concurrent programs, a task known to be difficult and error-prone when done manually. Our framework is based on abstract interpretation and can infer synchronization for infinite state programs. Given a program, a specification, and an abstraction, we infer synchronization that avoids all (abstract) interleavings that may violate the specification, but permits as many valid interleavings as possible. Combined with abstraction refinement, our framework can be viewed as a new approach for verification where both the program and the abstraction can be modified on-the-fly during the verification process. The ability to modify the program, and not only the abstraction, allows us to remove program interleavings not only when they are known to be invalid, but also when they cannot be verified using the given abstraction. We implemented a prototype of our approach using numerical abstractions and applied it to verify several interesting programs.

3.25 Specifications by Design

Jean Yang (MIT – Cambridge, US)

Joint work of Yang, Jean; Yessenov, Kuat; Solar-Lezama, Armando
Main reference J. Yang, K. Yessenov, A. Solar-Lezama, "A Language for Automatically Enforcing Privacy Policies," POPL'12.
URL http://people.csail.mit.edu/jeanyang/papers/popl088-yang.pdf

In this talk, I advocate thinking about language designs that encourage programmers to provide specifications for synthesis. I focus on the subject of privacy, a domain where the programmer currently bears the burden of ensuring that the application's behavior adheres to policies about where sensitive values may flow. Privacy policies are difficult to manage because their global nature requires coordinated reasoning and enforcement. To address this problem, we describe a programming model that makes the system responsible for ensuring adherence to privacy policies. The programming model has two components: 1) core programs describing functionality independent of privacy concerns and 2) declarative, decentralized policies controlling how sensitive values are disclosed. Each sensitive value encapsulates multiple views; policies describe which views are allowed based on the output context. The system is responsible for automatically ensuring that outputs are consistent with the policies. We have implemented this programming model in a new functional constraint language named Jeeves. In Jeeves, sensitive values are introduced as symbolic variables and policies correspond to constraints that are resolved at output channels. We have implemented Jeeves as a Scala library using an SMT solver as a model finder. I describe the Jeeves programming language and our experience using Jeeves to implement a conference management system. I encourage others to think about lifting abstractions in ways that facilitate the communication of specifications.

3.26 Program Repair Revisited

Christian von Essen (VERIMAG – Gières, FR)

Joint work of von Essen, Christian; Jobstmann, Barbara
Main reference C. von Essen, B. Jobstmann, “Program Repair Revisited,” Verimag Research Report TR-2012-4
URL http://www-verimag.imag.fr/Technical-Reports,264.html?lang=en&number=TR-2012-4

We present a new and flexible approach to repair reactive programs with respect to a specification. The specification is given in linear-temporal logic. Like in previous approaches, we require that a repaired program satisfies the specification and is syntactically close to the faulty program. In addition our approach also allows the user to ask for a program that is semantically close by enforcing that a specific subset of the correct traces is preserved. Our approach is based on synthesizing a program producing a set of traces that stays within a lower and an upper bound. We provide an algorithm to decide if a program is repairable with respect to our new notion and synthesize a repair if one exists. We analyze several ways to choose the set of traces to leave intact and show the boundaries they impose on repairability.

Participants

- Rajeev Alur
University of Pennsylvania, US
- Don Batory
University of Texas at Austin, US
- Rastislav Bodik
University of California – Berkeley, US
- Krzysztof Czarnecki
University of Waterloo, CA
- Jyotirmoy Deshmukh
University of Pennsylvania, US
- Laurent Doyen
ENS – Cachan, FR
- Bernd Finkbeiner
Universität des Saarlandes, DE
- Pierre Flener
Uppsala University, SE
- Franz Franchetti
Carnegie Mellon University – Pittsburgh, US
- Sumit Gulwani
Microsoft Res. – Redmond, US
- Amey Karkare
Indian Inst. of Technology – Kanpur, IN
- Hadas Kress-Gazit
Cornell University, US
- Rupak Majumdar
MPI for Software Systems – Kaiserslautern, DE
- Oded Maler
VERIMAG – Gières, FR
- Mark Marron
IMDEA Software – Madrid, ES
- Vu Minh Le
University of California – Davis, US
- Alon Mishne
Technion – Haifa, IL
- Jean-Noël Monette
Uppsala University, SE
- Georg Ofenbeck
ETH Zürich, CH
- Doron A. Peled
Bar-Ilan University – Ramat-Gan, IL
- Ruzica Piskac
MPI für Softwaresysteme – Saarbrücken, DE
- Nir Piterman
University of Leicester, GB
- Corneliu Popeea
TU München, DE
- Subhajit Roy
Indian Inst. of Technology – Kanpur, IN
- Roopsha Samanta
Univ. of Texas at Austin, US
- Sven Schewe
University of Liverpool, GB
- Sanjit A. Seshia
University of California – Berkeley, US
- Armando Solar-Lezama
MIT – Cambridge, US
- Saurabh Srivastava
University of California – Berkeley, US
- Philippe Suter
EPFL – Lausanne, CH
- Nathalie Sznajder
UPMC – Paris, FR
- Emina Torlak
University of California – Berkeley, US
- Stavros Tripakis
University of California – Berkeley, US
- Martin T. Vechev
ETH Zürich, CH
- Christian von Essen
VERIMAG – Gières, FR
- Eran Yahav
Technion – Haifa, IL
- Jean Yang
MIT – Cambridge, US