# 09501 Abstracts Collection
# Software Synthesis
## — Dagstuhl Seminar —

Rastislav Bodik,[1], Orna Kupferman,[2], Douglas R. Smith,[3] and Eran Yahav,[4]

[1] Univ. of California, Berkeley, USA
[2] Hebrew Univ., Israel
[3] Kestrel Institute, Palo Alto, USA
[4] IBM Research, New York, USA

**Abstract.** From 06.12.09 to 11.12.09, the Dagstuhl Seminar 09501 "Software Synthesis " was held in Schloss Dagstuhl – Leibniz Center for Informatics. During the seminar, several participants presented their current research, and ongoing work and open problems were discussed. Abstracts of the presentations given during the seminar as well as abstracts of seminar results and ideas are put together in this paper. The first section describes the seminar topics and goals in general. Links to extended abstracts or full papers are provided, if available.

**Keywords.** Software Synthesis, Verification, Theorem Proving, Program Analysis, Programming by Demonstration

## 09501 Executive Summary – Software Synthesis

Recent years have witnessed resurgence of interest in software synthesis, spurred by growing software complexity and enabled by advances in verification and decision procedures. This seminar brought together veterans of deductive synthesis as well as representatives of new synthesis efforts. Collectively, the seminar assembled expertise in diverse synthesis techniques and application areas,

The first half of the seminar focused on educating the participants in foundations and empirical results developed over the last three decades in the mostly isolated synthesis communities. The seminar started with tutorial talks on deductive synthesis, controller synthesis, inductive synthesis, and the use of decision procedures in program synthesis. The second half of the seminar led to a lot of discussion, boosted by talks on specific software synthesis problems.

The participants agreed that there are several reasons to actively explore synthesis now. First, software development, always non-trivial, is likely to become more complicated as a result of transition to multi-core processors. The hope is that we will synthesize at least the hard fragments of parallel programs. Second, deductive program verification and synthesis are intimately related; it seems promising to explore whether results in model checking and directed testing enable interesting synthesis. Third, by incorporating verification into synthesis we may be able to synthesize programs that are easier to verify than handwritten programs. Finally, the continuing Moore's Law may enable search powerful enough for synthesis of practical programs.

The seminar also led to identification of principles and open problems in benchmarking of software synthesis tools. In contrast to benchmarking of compilers and verifiers, experiments with synthesis must evaluate end-to-end benefits in programmer productivity; in particular, can the program be developed faster with the synthesizer than with a modern programming language? Short of performing a controlled user study, little can be said about the magnitude of these benefits. The situation is more favorable when comparing synthesis tools. The participants agreed that experiments reported in the literature must identify the knowledge that the user had to formalize in the domain theory that made the synthesis possible. It was also deemed important to identify the formalism used in expressing the domain knowledge.

General Conclusions from the Seminar. The participants found the seminar to be educational and inspiring. We believe this was because of the unusual breadth of participants as well as the format, which revolved around tutorial-style talks that brought the participating communities together.

The participants believed that the talks should be shared with graduate students, who are usually exposed in their courses only to a fraction of synthesis techniques. This observation led to organization of summer school on synthesis, which will be held in Dagstuhl in summer 2011.

The need to create a collection of diverse synthesis results also led to a special issue of the STTT journal of software synthesis, which is under preparation.

*Keywords:* Software Synthesis, Verification, Theorem Proving, Program Analysis, Programming by Demonstration

*Joint work of:* Bodik, Rastislav

## Angelic Programming

*Satish Chandra (IBM TJ Watson Research Center - Hawthorne, US)*

Angelic nondeterminism can play an important role in program development. It simplifies specifications, for example in deriving programs with a refinement calculus; it is the formal basis of regular expressions; and Floyd relied on it to concisely express backtracking algorithms such as N-queens.

We show that angelic nondeterminism is also useful during the development of deterministic programs. The semantics of our angelic operator are the same as Floyd's but we use it as a substitute for yet-to-be-written deterministic code; the final program is fully deterministic. The angelic operator divines a value that makes the program meet its specification, if possible. Because the operator is executable, it allows the programmer to test incomplete programs: if a program has no safe execution, it is already incorrect; if a program does have a safe execution, the execution may reveal an implementation strategy to the programmer.

We introduce refinement-based angelic programming, describe our embedding of angelic operators into Scala, report on our implementation with bounded model checking, and describe our experience with two case studies. In one of the studies, we use angelic operators to modularize the Deutsch-Schorr-Waite (DSW) algorithm. The modularization is performed with the notion of a parasitic stack, whose incomplete specification was instantiated for DSW with angelic nondeterminism.

*Joint work of:* Chandra, Satish; Rastislav Bodik; Joel Galenson; Doug Kimelman; Nicholas Tung; Shaon Barman; Casey Rodarmor

## Synthesizing Verifiers for Synthesized Code

*Ewen W. Denney (NASA - Moffett Field, US)*

Automated code generators are increasingly used in safety-critical applications, but since they are typically not qualified, the generated code must still be fully tested, reviewed, and certified. For mathematical and engineering software this requires reviewers to trace subtle details of textbook formulas and algorithms to the code, and to match requirements (e.g., involving physical concepts such as units or coordinate frames) not represented explicitly in models or code.

The AutoCert verification system identifies and verifies mathematical concepts in code, recovering verified traceability links between concepts, code, and verification conditions.

The verification is customized by domain knowledge represented as a set of schemas, which use patterns to describe code idioms and actions to construct annotations needed to certify matching code fragments.

We can raise the level of abstraction at which schemas are defined and generate lower-level schemas, that is, compile high-level domain knowledge into low-level verification knowledge. The schema compiler can itself therefore be seen as a domain-specific code generator.

*Keywords:* Synthesis, verification

## CEGAR for Synthesis

*Bernd Finkbeiner (Universität des Saarlandes, DE)*

Counterexample-guided abstraction refinement (CEGAR) is used in automated software analysis to find suitable finite-state abstractions of infinite-state systems. In this talk, we extend CEGAR to games with incomplete information as they commonly occur in software synthesis.

The challenge is that, under incomplete information, one must carefully account for the knowledge available to the player: the strategy must not depend on information the player cannot see. We propose an abstraction refinement mechanism for games with incomplete information that incorporates the approximation of the players' moves into a knowledge-based subset construction on the abstract state space. This abstraction results in a perfect-information game over a finite graph. The concretizability of abstract strategies can be encoded as the satisfiability of strategy-tree formulas. Based on this encoding, we present an interpolation-based approach for selecting new predicates. Joint work with Rayna Dimitrova.

## Dimensions in Program Synthesis

*Sumit Gulwani (Microsoft Research - Redmond, US)*

Program Synthesis, which is the task of discovering programs that realize user intent, can be useful in several scenarios: enabling people with no programming background to develop utility programs, helping regular programmers automatically discover tricky/mundane details, program understanding, discovery of new algorithms, and even teaching.

This paper describes three key dimensions in program synthesis: expression of user intent, space of programs over which to search, and the search technique. These concepts are illustrated by brief description of various program synthesis projects that target synthesis of a wide variety of programs such as standard undergraduate text- book algorithms (e.g., sorting, dynamic programming), program inverses (e.g., decoders, deserializers), bitvector manipulation routines, deobfuscated programs, graph algorithms, text-manipulating routines, mutual exclusion algorithms, etc.

## Component based Synthesis

*Sumit Gulwani (Microsoft Research - Redmond, US)*

We present a novel approach to automatic synthesis of loop-free programs. The approach is based on a combination of oracle-guided learning from examples, and constraint-based synthesis from components using satisfiability modulo theories (SMT) solvers. Our approach is suitable for many applications, including as an aid to program understanding tasks such as deobfuscating malware. We demonstrate the efficiency and effectiveness of our approach by synthesizing bit-manipulating programs and deobfuscating programs.

## Tutorial: Synthesis and Games

*Barbara Jobstmann (VERIMAG - Gières, FR)*

In the first part, I will give a basic introduction to infinite game theory and its relationship to synthesis from temporal logic formulas. The second part gives a summary of how we used these techniques to repair finite-state programs and construct hardware designs specified by a set of Linear Temporal Logic formulas.

*Keywords:* Games, synthesis, program repair

## Synthesizing Robust Systems

*Barbara Jobstmann (VERIMAG - Gières, FR)*

Many specifications include assumptions on the environment. If the environment satisfies the assumptions then a correct system reacts as intended. However, when the environment deviates from its expected behavior, a correct system can behave arbitrarily.

In this talk, I will discuss how to synthesize robust systems that degrade gracefully, i.e., a small number of environment failures should induce a small number of system failures.

*Keywords:* Robust systems, ratio games, synthesis

## Synthesis of Communicating Automata from MSCs

*Joost-Pieter Katoen (RWTH Aachen, DE)*

This paper is concerned with bridging the gap between requirements and distributed systems. Requirements are defined as basic message sequence charts (MSCs) specifying positive and negative scenarios. Communicating finite-state machines (CFMs), i.e., finite automata that communicate via FIFO buffers, act as system realizations. The key contribution is a generalization of AngluinÃŋs learning algorithm for synthesizing CFMs from MSCs.

This approach is exact - the resulting CFM precisely accepts the set of positive scenarions and rejects all negative ones' and yields fully asynchronous implementations. The paper investigates for which classes of MSC languages CFMs can be learned, presents an optimization technique for learning partial orders, and provides substantial empirical evidence indicating the practical feasibility of the approach.

## Efficient Synthesis of Asynchronous Systems

*Uri Klein (Courant Institute - New York, US)*

We investigated a development process for reactive programs, in which the program is automatically generated (synthesized) from a high-level temporal (property) specification. The method is based on previous results that proposed a similar synthesis method for the automatic construction of hardware designs from their temporal specifications. Thus, our work can be viewed as a generalization of existing methods for the synthesis of synchronous reactive systems into the synthesis of asynchronous systems.

In the synchronous case it was possible to identify a restricted yet expressive subclass of formulas and present an algorithm that solves the synthesis problem for these restricted specifications in polynomial time. Here, due to a possibly exponential increase in complexity of the synthesis problem, the results are less definitive in the sense that we can offer some heuristics that may provide polynomial-time solutions only in some of the cases. The approach taken here is to extract from the specification two new specifications that, in some sense, provide an upper and a lower bound to the specification. The upper bound (over-approximation) specification could be used to determine that the original specification is unrealizable. On the other hand, the lower bound (under-approximation) could be used in order to determine that the original specification is realizable, and to actually produce an implementation of this specification. As mentioned above, the realizability analysis of these two approximations can be performed in polynomial time for the case that the original asynchronous specification falls into the restricted class.

Joint work with Amir Pnueli.

## Synthesis Procedures

*Viktor Kuncak (EPFL - Lausanne, CH)*

Synthesis of program fragments from specifications can make programs easier to write and easier to reason about. To integrate synthesis into programming languages, synthesis algorithms should behave in a predictable way: they should succeed for a simple-to-describe class of specifications. They should also support unbounded data types such as numbers and data structures. We show how to generalize decision procedures into predictable and complete synthesis procedures. Such synthesis procedures are guaranteed to find code that satisfies the specification if such code exists. Moreover, we identify conditions under which synthesis will statically decide whether the solution is guaranteed to exist, and whether it is unique. We demonstrate our approach by extending decision procedures for linear arithmetic and set data structures into synthesis procedures, and establishing results on the size and the efficiency of the synthesized code. We show that such procedures are useful as a language extension with implicit value definitions and advanced pattern- matching constructs. We show how to extend the Scala compiler to support such definitions. Our constructs provide the benefits of synthesis to programmers, without requiring them to learn fundamentally new concepts or give up a deterministic program execution model.

*Keywords:* Decision procedure, infinite-state synthesis, quantifier elimination

*Joint work of:* Mayer, Mikael; Suter, Philippe; Piskac, Ruzica; Kuncak, Viktor

## Synthesis as Learning Automata from Examples

*Martin Leucker (TU München, DE)*

In this tutorial presentation, we recall two automata learning algorithms and show their applications in the area of verification and synthesis.

*Keywords:* Learning, synthesis, verification

## Systematic Program Design: From Clarity to Efficiency

*Yanhong Annie Liu (SUNY - Stony Brook, US)*

Two major concerns of study rest at the center of computer science: what to compute, and how to compute efficiently. Problem solving involves going from clear specifications for the "what" to efficient implementations for the "how". This is challenging because clear specifications usually correspond to straightforward implementations, not at all efficient, while efficient implementations are usually difficult to understand, not at all clear.

This talk gives an overview of a general and systematic method for transforming clear specifications into efficient implementations. The method has three steps: (1) iterate—determine a minimum increment to be taken repeatedly, (2) incrementalize—maintain appropriate values incrementally over the repeated steps, and (3) implement—design data structures for the values maintained. We will illustrate the method through examples, taken from problems in hardware design and image processing expressed using loops and arrays, in query processing and access control expressed using set operations, in sequence processing and math puzzles expressed using recursive functions, in program analysis and trust management expressed using logic rules, and in building software components expressed using objects. Finally, we summarize our ongoing projects on a number of fronts.

*Keywords:* Incrementalization, program optimization, program transformation

## Synchronous Reactive Synthesis From Linear Temporal Logic Specifications

*Nir Piterman (Imperial College London, GB)*

In this talk we are interested in synchronous reactive synthesis from Linear Temporal Logic specifications. Classical solutions to synthesis use either two player games or tree automata.

The classical solution to synthesis requires the usage of deterministic automata. This solution is 2EXPTIME-complete, is quite complicated, and does not work well in practice.

We suggest a syntactic approach that restricts the kind of properties users are allowed to write.

We claim that this approach is general enough and can be extended to cover most properties written in practice.

The main advantage of our approach is that it is tailored to the use of BDDs and uses the structure of given properties to handle them more efficiently.

We discuss how to extend our approach to handle more general properties and mention future direction.

*Keywords:* Synthesis, LTL

*Joint work of:* Piterman, Nir; Pnueli, Amir; Sa'ar Yaniv

*Full Paper:*
http://www.doc.ic.ac.uk/~npiterma/publications/2006/PPS06.pdf

*See also:* N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. 2006. In Proc. 7th International Conference on Verification, Model Checking and Abstract Interpretation,volume 3855 of Lecture Notes in Computer Science, pages 364-380. Â©Springer-Verlag.

## Synthesizing Hardware from Sketches

*Andreas Raabe (TU München, DE)*

Due to the on-going micro-miniaturization in chip production, hardware development faces new challenges. First, the designer productivity grows slower than the number of transistors per chip.

Second, the rate of innovation has increased to a point where only the first to the market makes a profit. Hardware synthesis has been successfully used to improve designer productivity, but in general it results in designs considerably slower and bigger than hand-coded hardware descriptions.

In high-level synthesis, the synthesizer acts as a smart compiler, translating high-level specifications into low-level designs. We recognize that the designer is best equipped to create smart low-level designs. Therefore, we allow the designer

to produce intricate lowlevel designs. However, low-level designs are tedious to write. To this end, sketching will allow the designer to leave tricky details unspecified. The incomplete design description is called a sketch.

The sketch will be completed by the synthesizer to meet a separate executable specification.

*Keywords:* Hardware Development, Sketching

## From tests to proofs through a simple trick

*Andrey Rybalchenko (MPI für Software Systeme - Saarbrücken, DE)*

We describe the design and implementation of an automatic invariant generator for imperative programs that leverages dynamic information collected from concrete and symbolic execution.

While automatic invariant generation through constraint solving has been extensively studied from a theoretical viewpoint as a classical means of program verification, in practice existing tools do not scale even to moderately sized programs.

This is because the constraints that need to be solved even for small programs are already too difficult for the underlying (non-linear) constraint solving engines.

To overcome this obstacle, we propose to strengthen static constraint generation with information obtained from static abstract interpretation and dynamic execution of the program.

The strengthening comes in the form of additional linear constraints that trigger a series of simplifications in the solver, and make solving more scalable.

We demonstrate the practical applicability of the approach by an experimental evaluation on a collection of challenging benchmark programs and comparisons with related tools based on abstract interpretation and software model checking.

Joint work with Ashutosh Gupta and Rupak Majumdar

*Keywords:* Invariant Synthesis

*Full Paper:*
http://www.springerlink.com/content/h3t77333468hk7m5/

## Software Synthesis is Hard – and Simple

*Sven Schewe (University of Liverpool, GB)*

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.

*Keywords:* Synthesis, Temporal Logics

*Full Paper:* http://drops.dagstuhl.de/opus/volltexte/2010/2670

## Data Mining of Air Traffic Track Data for NGATS with the AutoBayes Synthesis System

*Johann M. Schumann (NASA - Moffett Field, US)*

The Next Generation Air Traffic Control System (NGATS) heavily relies on the accurate prediction of aircraft trajectories. Tasks like the detection of separation conflicts or planning fuel-efficient and safe descents require the system to predict position, altitude, and speed of an aircraft for up to appr. 20 minutes ahead, based upon the current position and weather data. Simulation of a physical model of the aircraft forms the core of the prediction. In practice, however, there is a number of unknowns to be taken into account. Typically, weight of a specific aircraft as well as other performance data or procedural specifics are not available, but are essential for an accurate prediction.

In this abstract, we describe how the AutoBayes tool can be used to extract parameters of interest from actual track data. Track data are recordings of actual air traffic in 12 second intervals, which contain position, altitude, ground speed, heading, among other data for each aircraft in a specific airspace sector.

Typically, a 24hour recording around a major airport contains thousands of aircraft tracks.

We have been using such data to determine several aircraft parameters and track characteristics, most notably, clustering of different trajectory types, characteristics of CDA (Constant Descent Approaches), and determination of the CAS-mach transition point during ascent. In this abstract, we focus on the CAS-mach transition. When an aircraft climbs toward cruise altitude, it usually starts its climb with a constant (calibrated) airspeed (CAS), measured in knots. At a certain point, the flight management system (FMS) on-board the aircraft switches over to a climb regime, where the mach number (relative speed

with respect to the speed of sound) is kept constant. It is obvious that the prediction accuracy can be improved if all parameters of the switch are known; however usually these not available.

We are using the AutoBayes tool for the change-point estimation on large sets of track data. AutoBayes is a tool for the automatic generation of efficient data analysis algorithms (in C/C++), given a compact statistical specification.

Internally, AutoBayes constructs a Bayesian network and evaluates the required probabilistic expressions symbolically as far as possible. Using a schema-based program synthesis approach, AutoBayes can generate complicated yet highly customized algorithms fully automatically. For this analysis, we have used several variants of change-point detection and clustering of mixture models. With a simple Matlab interface, we have been able to identify suitable ascent trajectories and to obtain the transition points for different types of aircraft. This analysis is a first step toward using AutoBayes statistical models for the analysis of aircraft track data for analysis and verification/validation purposes.

*Keywords:* Synthesis, Data analysis algorithm, data mining

## Mechanized Algorithm Design

*Douglas R. Smith (Kestrel Institute, US)*

This talk is a tutorial/overview of mechanized algorithm design, as supported by the KIDS, Specware, and Planware systems at Kestrel Institute. Starting from a logical specification of an algorithmic problem, a typical synthesis is a sequence of refinement steps: algorithm design, followed by various optimizations, then refinement of abstract datatypes to concrete implementations. Our emphasis is on composing representations of reusable design knowledge in the form of algorithm theories, datatype refinements, and program optimization tactics. Applications to sophisticated scheduling problems and the generation of fast SAT solvers are discussed. The same abstract design knowledge was used to generate schedulers and SAT solvers.

*Keywords:* Formal specification, algorithm theories, automated synthesis

## New directions in Computer Supported Programming

*Armando Solar-Lezama (MIT - Cambridge, US)*

This talk will describe some of the new research directions currently being pursued by the computer supported programming group at MIT. The talk will highlight how our current research is attacking three important sources of programming difficulty: low-level algorithmic details, scale, and coping with the unexpected.

Sketching was developed to allow programmers to synthesize the low-level details of the algorithm while retaining control over the final form of the implementation. In the first part of the talk, I will describe our recent results in extending sketch-based synthesis to the domains of numerical control applications and complex data-structure manipulations. Both of these domains are a challenge for the standard CEGIS based sketch synthesis algorithm. I will show how numerical programs can be synthesized by using standard numerical methods with the aid of a new form of analysis called "smooth interpretation". For data structures, I will present some our recent work on combining the standard CEGIS algorithm with abstract interpretation to achieve improved scalability.

The second part of the talk will describe our recent efforts to use data driven program analysis to tackle the challenges of adding functionality to very large applications. These applications pose a challenge for programming tools because of their extreme scale, but by collecting large amounts of program behavior data, we can make up for the shortcomings of static analysis and help the programmer understand such large applications.

Finally, the third part will describe our recent work on "declarative program hardening" and its application to data processing applications. Declarative program hardening allows programmers to focus on the common case behavior of their programs, while separately asserting facts that the program can use to cope with exceptional situations. In the context of data processing, this technique allows programmers to cope with missing or corrupted data in a clean and robust way.

## Program Synthesis using SMT Solvers

*Saurabh Srivastava (University of Maryland - College Park, US)*

In this talk, we describe two approaches to program synthesis: one that is inspired by verification, called proof-theoretic synthesis, and one that is inspired by testing, called path-based inductive synthesis (PINS). Proof-theoretic synthesis is attractive in that it intrinsically synthesizes the proof (invariants, ranking functions) alongside the program; while PINS has the benefit of being able to handle more complex programs as it does not attempt to infer the formal proof.

The insight behind proof-theoretic synthesis is to interpret program synthesis as *generalized program verification*, which allows us to directly use verification tools, such as the ones we built in previous work, as synthesizers. The approach requires the user to state the required functional specification and a template of the desired form of the program (looping structure, stack space etc). Using this, our synthesis algorithm generates constraints over unknown statements and guards, and additionally proof terms—inductive invariants and ranking functions. The constraints enforce three kinds of requirements: partial correctness, loop termination, and well-formedness of the program. It then synthesizes the program (in “one shot”) by solving the constraints using an off-the-shelf verifier with particular properties.

We have used this approach to synthesize programs in three different domains: arithmetic, sorting, and dynamic programming. Using verification tools, that we built in prior work we are able to synthesize programs for complicated arithmetic algorithms including Strassen's matrix multiplication and Bresenham's line drawing; several sorting algorithms; and several dynamic programming algorithms, in very reasonable time.

Path-based inductive synthesis (PINS) is an iterative approach to synthesis inspired by the philosophy behind testing—that a set of carefully chosen paths through the program give sufficient indications of its functionality. In this approach, we generate safety and termination constraints *over paths* and solve the resulting constraints to synthesize the program. These constraints again have nested quantification and we employ the solution techniques, that we developed for verification, for solving such constraints.

We have used this approach to synthesize inverses for complicated programs such as stream compressors, formatters, arithmetic examples, and additionally, for synthesizing a TFTP client from its server.

## Synthesis of Concurrent Algorithms

*Martin T. Vechev (IBM TJ Watson Research Center - Hawthorne, US)*

Practical and efficient concurrent algorithms are difficult to construct, verify and modify.

Concurrent Algorithms in the literature are often optimized for a specific setting, making it hard to separate the algorithmic insights from implementation details.

The goal of this work is to systematically construct concurrent algorithms starting from their sequential implementation. Towards that goal, we follow a construction process that combines manual steps corresponding to high-level insights with automatic exploration of implementation details. To assist us in this process, we built a new tool that quickly explores large spaces of algorithms and uses a checking or a verification procedure for ensuring the safety of the algorithms.

Starting from a sequential implementation and assisted by the tool, we present the steps that we used to derive various highly concurrent algorithms. Among these algorithms is a new fine grained set data structure that provides a wait-free contains operation, and uses only the compare-and-swap (CAS) primitive for synchronization.

*Keywords:* Concurrency, synthesis, verification, algorithms

## Recursive Plans and Imperative Programs Revisited: Deductive Synthesis

*Richard Waldinger (SRI - Menlo Park, US)*

The formulation of plans is viewed as a problem of deductive inference. A plan that meets a specified goal is extracted from a proof of an appropriate theorem, where the proof is discovered by an automatic theorem prover. The special focus of this talk is the formation of plans that require repetitive actions, which are represented by recursion. Such a plan results from the use of the mathematical induction principle during the proof.

We have considered examples in workflow, the Levesque tree-chopping problem, classical blocks-world and robotic problems, and some samples of intelligent bird behavior. Imperative programs, which have side effects such as the alteration of data structures, can be treated in the same way as plans. The talk reports on the results of experiments done with an automatic theorem prover (StickelÃŋs SNARK). The work is motivated by potential applications in molecular biology and biomedical research and business workflow formation.

*Keywords:* Planning, program synthesis, theorem proving, mathematical induction, well-founded induction, SNARK

## Abstraction-Guided Synthesis of Synchronization

*Eran Yahav (IBM TJ Watson Research Center - Hawthorne, US)*

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.

*Keywords:* Synthesis, concurrent programs, abstract interpretation

*Joint work of:* Vechev, Martin; Yahav, Eran; Yorsh, Greta

## Inferring Synchronization under Limited Observability

*Greta Yorsh (IBM TJ Watson Research Center - Hawthorne, US)*

This paper addresses the problem of automatically inferring synchronization for concurrent programs. Given a program and a specification, we infer synchronization that avoids all interleavings violating the specification, but permits as many valid interleavings as possible. We let the user specify an upper bound on the cost of synchronization, which may limit the observability - what observations on program state can be made by the synchronization code. We present an algorithm that infers, under certain conditions, the maximally permissive synchronization for a given cost.

We implemented a prototype of our approach and applied it to infer synchronization in a number of small programs.

*Joint work of:* Martin Vechev, Eran Yahav, Greta Yorsh