Int J Softw Tools Technol Transfer (2013) 15:397–411
DOI 10.1007/s10009-013-0287-9

INTRODUCTION

Algorithmic program synthesis: introduction

Rastislav Bodik · Barbara Jobstmann

Published online: 27 August 2013

Abstract Program synthesis is a process of producing an executable program from a specification. Algorithmic synthesis produces the program automatically, without an intervention from an expert. While classical compilation falls under the definition of algorithmic program synthesis, with the source program being the specification, the synthesis literature is typically concerned with producing programs that cannot be (easily) obtained with the deterministic transformations of a compiler. To this end, synthesis algorithms often perform a search, either in a space of candidate programs or in a space of transformations that might be composed to transform the specification into a desired program. In this introduction to the special journal issue, we survey the history of algorithmic program synthesis and introduce the contributed articles. We divide the field into *reactive synthesis*, which is concerned with automata-theoretic techniques for controllers that handle an infinite stream of requests, and *functional synthesis*, which produces programs consuming finite input. Contributed articles are divided analogously. We also provide pointers to synthesis work outside these categories and list many applications of synthesis.

Keywords Program synthesis · Controller Synthesis · Formal Methods · Specifications of Program Correctness

R. Bodik (✉)
Department of Electrical Engineering and Computer Sciences, University of California, #1776, Berkeley, CA 94720-1776, USA
e-mail: bodik@cs.berkeley.edu

B. Jobstmann
École Polytechnique Fédérale de Lausanne, Station 14, 1015 Lausanne, Switzerland

B. Jobstmann
CNRS-VERIMAG, Centre Equation, 2, Avenue de Vignate, 38610 Gières, France

1 Overview

1.1 Background

Since the early days of computer science, we have aimed to verify "correctness" of a computer system by proving its adherence to a logical specification, a mathematical description of the desired behavior. Synthesis goes one step further: it aims to construct or derive a program or a digital design from that logical specification. A fully automated solution to the synthesis problem would augment the current programming style, which is mostly operational, with a declarative, logical style. Stating the desired properties of the system, rather than elaborating how these properties are to be achieved, may be particularly beneficial for complex systems, such as those with concurrency.

While the theoretical basis for synthesis from logical specification has been known for decades, only recently has a rather broad class of synthesis systems reached the level of practical applications. These techniques share a high degree of automation afforded by advances in decision procedures and static program analysis developed in the context of verification and model checking. Additional advances stem from judicious restrictions to a predefined set of programs and from assuming a particular type of specification. For instance, two decision procedures [109,144] for realizability of Linear-Time Temporal Logic (LTL) have led to the development of new algorithms and tools for synthesis of finite-state programs from LTL specifications [15,48,86,87,106,136,158,170,172], some able to synthesize small but realistic industrial hardware modules [16,17,61]. Other techniques seek to simplify specific programming tasks by adding new programming constructs, e.g., synchronization mechanisms for concurrent data structures [175,195] or deadlock avoidance constructs [198,199], and by allowing implicit computation of values and advanced pattern matching [103]. Synthesis techniques have also been used to correct [88] or optimize [176] existing programs and to build programs from a fixed set of library components or program statements [85,177,174,180]. Synthesis of spreadsheet macros from user demonstrations has been incorporated in Excel 2013 [66].

1.2 Goal of this special issue

We have collected recent work in *algorithmic program synthesis* to provide a starting point for researchers and practitioners interested in this area. The contributions cover two main areas of program synthesis: synthesis of controllers for reactive systems and synthesis of functional and imperative programs. The former are control-oriented while the latter are data-oriented.

Control-oriented techniques assume finite-state data structures but can generate non-terminating reactive programs [127], i.e., programs that accept an infinite stream of requests. Input specifications describe the desired sequences of behaviors using a temporal logic (e.g., LTL). These techniques are based on automata theory on infinite words and automata-based game theory. Typically, reactive programs are used to model digital designs.

Data-oriented techniques assume a bounded (but unknown) length of the program but can cope with infinite-state data structures. They start from an input–output relation given by a formula in a realizable logic, e.g., integer arithmetic or propositional logic, and construct source code satisfying the specification. In some cases, the class of generated program can be restricted by the set of available statements or an initial partial program.

Due to the nature of these two classes of programs—controllers carry out an infinite request–response dialog with the environment using a finite-state controller, while imperative programs read the input only once but are unconstrained in their internal structure—, the algorithms for synthesis of these programs rely on radically different concepts. Yet, the respective communities have recently reached for similar techniques, such as exploiting domain knowledge, refinement, bounded synthesis, and constraint solvers. We hope that this special issue will help transfer ideas between the two communities.

In Sect. 2, we give background and history of controller synthesis. Section 3 does the same for functional synthesis. Section 4 lists synthesis work that does not fit either of the two categories. Section 5 describes applications of synthesis and Sect. 6 summarizes contributions of this special issue.

2 Reactive synthesis

This section focuses on *Reactive Synthesis*, which aims to automatically construct a reactive system from a formal specification. The specification is given in Linear-Temporal Logic (LTL) and describes logical properties of the system. For simplicity, we will refer to this form of high-level synthesis as *LTL synthesis*, but emphasize that it should not be confused with the synthesis of a gate-level description from VERILOG, RTL code, or from a high-level behavioral description.

2.1 Motivation

Automatically constructing a reactive system from a logical specification has been an ambitious dream in computer science for about half a century [30]. The obvious benefit is that one only has to give a list of desired behaviors and a synthesis tool comes up with a finite-state model that takes into account all desired properties. For systems that have no further constraints (e.g., on timing or space consumption), a synthesis tool would completely avoid hand-coding. A less ambitious benefit is the possibility to turn specifications into rapid prototypes. These functional prototypes could be used for early test integration. They would allow one to “simulate the specification.” Most developers rely on simulation to check if the constructed system meets their intent. In the hardware community, it is believed that formal verification can only be integrated in the mainstream design flow if it looks and feels like simulation [8]. A synthesis tool could thus provide a simulation-like experience for a formal specification, which would help people in understanding and writing a formal specification (since faults in the specification become apparent immediately). Synthesis is a good method for debugging a specification, a task that will gain importance as formal specification begins to be used as the basis for manual implementation.

Although reactive synthesis is solved in theory (see Sect. 2.2), synthesis tools have not gain acceptance in practice, for several reasons. One is that synthesis of LTL properties is 2EXPTIME-complete [152]. This reason alone should not prevent one from implementing the synthesis approach because this bound afflicts also a manually implemented system. This is because the bound is a lower bound, as shown in [152], so there are specifications for which the smallest correct system is doubly exponential in the size of the specification. Thus, the worst-case complexity of verifying the specification on a manual implementation is also 2EXPTIME in terms of the (full) specification. Another reason is that the classical solution to synthesis is not compositional and therefore does not reflect the usually iterative process of writing a complete specification. The lack of compositionality represents a serious obstacle because writing formal specifications is known to be difficult. Researchers follow three main ideas to cope with the complexity and usability of LTL synthesis problem: (i) **bounding** the size of the **generated systems**, (ii) **specialized algorithms** for subsets of LTL and way to **combined** these algorithms, and (iii) **partial program synthesis** to reduce the need for complete specifications. The papers in this issue describe these directions.

2.2 History of reactive synthesis

Synthesis aims to transform a specification into a system that is guaranteed to satisfy the specification. The theory behind synthesis of reactive systems is well established and goes back to Church, who stated the *Synthesis Problem* [30] using different fragments of *restricted recursive arithmetic* (S1S) as specification. Nowadays this problem is also know as *Realizability or Church's Problem* and is defined as follows: Given a relation $R \subseteq (2^I)^\omega \times (2^O)^\omega$ defined by restricted recursive arithmetic or another logic (e.g., LTL), we search for a function $f : (2^I)^* \rightarrow 2^O$ that generates for all sequences $x = x_0, x_1, x_2, ... \in (2^I)^\omega$ a sequence $y = y_0, y_1, y_2, ... \in (2^O)^\omega$ with $y_i = f(x_0, x_1, ..., x_{i-1})$ for $i > 0$ such that $R(x, y)$ holds. We can view I and O as the sets of input and output signals of a reactive system. The relation R can be seen as a linear specification including all pairs of input and output sequences that define the correct behavior of the system. The function f (called *strategy*) then maps every possible input sequence to a correct output sequence and represents a correct system.

In the following years, Büchi and Landweber [25] and Rabin [149] presented independent solutions to Church's problem. The first is based on infinite game theory, while the latter uses tree automata. It took nearly 10 years until researchers discovered the similarities between games and tree automata [70].

Specifying the behavior of reactive systems in S1S is cumbersome. This spurred an urgent need for new specification languages. A very successful proposal is that of *temporal logic* [45,146], which is now widely used in the formal verification community and provides the basis for commercial specifications languages as Property Specification Language (PSL) or System Verilog Assertions (SVA).

Essentially, temporal logic comes in two flavors: *linear* and *branching time*. Computation Tree Logic (CTL) [45] is a branching time logic. Given a CTL formula and a design, we can check efficiently if the design fulfills the formula. However, the restricted syntax of CTL limits the expressive power and makes writing specifications in CTL rather complicated. Specifying is easier in LTL [126,146], which is also more suitable for compositional reasoning [110,191]. In theory, the model checking problem for LTL is PSPACE-complete [167], but in practice (cf. [46,178]), it takes exponential time and space in the size of the LTL formula assuming $P \neq$ PSPACE.

The introduction of temporal logic gave rise to further developments in the area of synthesis of reactive systems. Emerson and Clarke [45] and Manna and Wolper [128] considered the problem for temporal specifications given in CTL and LTL, respectively. Both concluded that if a specification φ is satisfiable, we can construct a system that adheres to the specification using the model that satisfies φ. Due to the reduction to satisfiability, the approaches are limited to constructing *closed systems*, which lead to systems that are only guaranteed to work correctly in cooperative environments (environments that help to satisfy φ).

In the late 1980s, Pnueli and Rosner [147] reconsidered the topic for LTL and provided a solution for constructing *open systems* (reactive systems). The key observation (also observed in [149]) is that even though the specification can be represented as infinite sequences (words) over the input and output signals, the solution to the synthesis problem is an infinite tree. Furthermore, Rosner proved that synthesis of LTL properties is 2EXPTIME-complete [152]. The first exponent derives from the translation of the LTL formula into a non-deterministic Büchi automaton. The second exponent is due to the determinization of the automaton. In theory LTL synthesis was classified as solved but it was said to be hopelessly intractable and so the topic was dropped for a decade.

At the same time, Ramadge and Wonham [150] introduced the problem of *controller synthesis*, which deals with constructing a *controller* for a *plant*. They considered specification of the form $\textsf{always}\ p$ and proved that the controller can be synthesized in linear time for such specifications.

In order to overcome the complexity issues of LTL synthesis, people started to concentrate on subsets of the language, for e.g., Maidl [123] and Alur and La Torre [2] identified subsets of LTL for which deterministic automaton of less than doubly exponential size can be constructed. Wallmeier et al. [197] provided an efficient symbolic algorithm to synthesize *request–response* specification, which are of the form $\textsf{always}\,(p_i \rightarrow \textsf{eventually}\ q_i)$ for $i \in \{0..n\}$. Harding et al. [73] observed that by leaving out the determinization step, they obtain an efficient but incomplete symbolic algorithm. Jobstmann et al. [88] made the same observations independently in the context of using LTL synthesis to repair finite-state systems.

Recently, Pnueli et al. [144] proposed an efficient symbolic algorithm to automatically synthesize designs from LTL formulas belonging to the class of *generalized reactivity* of rank 1 (GR(1)) in time N^3, where N is the size of the state space of the design. The class GR(1) covers the vast majority of properties used in practice. GR(1) formulas have the form $(\bigwedge_{i\in\{0\cdots n\}} \textsf{always}\ \textsf{eventually}\,(E_i)) \rightarrow$

($\bigwedge_{j\in\{0\cdots m\}}$ $\textsf{always eventually}$ (S_j)), where E_i and S_j are Boolean formulas over atomic propositions representing the signals of the system. Most commonly used specifications can be transformed into this form using deterministic monitors.

Work concentrating on full LTL is sparse and has not been pursued for a long time. A major issue in Rosner's solution to LTL synthesis is the construction of a deterministic automaton for the given formula, which includes constructing a non-deterministic Büchi automaton and determinizing it. The first translation for LTL to non-deterministic Büchi automaton has been proposed by [202], and since then translators from LTL to automaton have improved a lot (cf. [33,59,178]). Unfortunately, constructions to determinize (arbitrary) Büchi automaton did not make such progress. In 1988, Safra [154] was the first to provide a determinization construction that was asymptotical optimal. The lower bound to determinization [118] was extended from the lower bound of [133] to the complementation of non-deterministic Büchi automata. (Kupferman and Vardi [107] showed a doubly exponential lower bound when one starts from LTL.) Later, Muller and Schupp [139] and Klarlund [93] provided different constructions that also match the lower bound. These algorithms turned out to be quite resistant to efficient implementations [80,94,160]. So their implementations can determinize automaton with approximately ten states and Rosner's solution to the Synthesis Problem was never implemented.

In 2005, Kupferman and Vardi [109] proposed an alternative solution that goes through universal co-Büchi and weak alternating tree automata. In contrast to previous approaches, it avoids Safra's determinization constructions. This approach was optimized by Schewe and Finkbeiner and implemented by Jobstmann and Bloem [86] and Filiot et al. [49].

Kupferman et al. [106] proposed a compositional version of the Safraless approach, which is based on generalized universal co-Büchi automata. In 2006, Piterman proposed improvements to Safra's construction that lead to better complexity bounds and deterministic Parity Automata [143]. In [79], Henzinger and Piterman introduced non-deterministic automata that are good for games (GFG). Those automata fairly simulate their deterministic equivalent and can be used to solve the synthesis problem. Henzinger and Piterman provided a simple algorithm to construct GFG automata from non-deterministic Büchi automata. The construction is a replacement for a determinization construction and so it has the same worst-case complexity. However, since it can be implemented symbolically, it is expected to perform better in practice. In [97], Kretínský and Esparza presented a direct translation of the (F,G)-fragment of LTL into deterministic ω-automata that avoids the known determinization procedures.

3 Functional synthesis

A defining characteristic of a synthesizer is the *candidate space*, which is the set of programs that the synthesizer considers when deriving (or searching for) a program that meets the specification. The synthesizers of the previous section considered the space of finite-state programs. The candidate spaces of this section will be more expressive, ranging from relational queries to arbitrary programs in general-purpose languages. This section divides synthesizers on the basis of whether they define the candidate space semantically (with axioms) or syntactically (with grammars). This distinction impacts both the expressiveness and the algorithmic foundations of the synthesizer. (Additional characterizations of the synthesis landscape can be found in [65,161].)

- Axiomatic synthesizers are equipped with a set of axioms in the form of an equational domain algebra or semantics-preserving rewrite rules. The desired program is obtained by axiom-driven derivation from the specification. Since the axioms are semantics-preserving, the resulting program is correct by construction. An advantage of axiomatic synthesizers is efficiency; they search only a space of correct programs (since only correct programs can be derived with given axioms). A disadvantage is that a complete behavioral specification is typically required as the starting point for the derivation; it is nontrivial to incorporate partial specifications, for example, in the form of input–output pairs. In terms of expressiveness, axiomatic synthesizers can synthesize only programs derivable with the set of axioms.
- Syntactically defined synthesizers consider candidates from a language defined, typically, by a context-free grammar. In practice, the syntactic pattern is defined by a programming language construct, such as a partial program, which is a parameterizable template program. Naturally, most candidate programs do not meet the specification, so the synthesis problem boils down to searching the candidate space for a correct program. To judge correctness, the synthesis algorithm employs a program checker. The use of a checker facilitates flexible specifications, which could combine safety properties with input–output pairs. In terms of expressiveness, syntactically defined can synthesize those programs that the checker can validate. The correctness assurance also rests on the checker, whose strength can vary from a full verifier to a dynamic program tester.

3.1 Axiomatic synthesizers

Problem domains with well-developed mathematical foundations, such as relational algebra [31] or linear filters [190], are amenable to axiomatization needed for synthesis. This subsection organizes existing work on the basis of the problem domain and discusses the nature of the domain algebra and the synthesis problems solved with the algebra.

Linear filters FFTW [58] and Spiral [148] develop schema-guided synthesizers for linear filters such as Discrete Fourier Transform (DFT). Efficient algorithms for linear transforms work by recursively decomposing the transform into transforms of smaller sizes. The schema defines the divide-and-conquer decomposition, giving rise to a particular linear-filter algorithm, such as the Cooley–Tukey algorithm for FFT [157]. The FFTW system optimizes the base case of the decomposition with DFT-specific tree rewrite and register allocation rules, discovering novel algorithms. Spiral uses a library of divide-and-conquer rules obtained by formalizing the published transform algorithms; an implementation is synthesized by selecting a suitable algorithm on each level of decomposition, guided by an empirical search. Spiral's operator language, based on linear algebra, uniformly supports not only linear-filter algorithm rules but also parallel-hardware compilation rules and recently also non-linear algorithms [56]. The StreamBit [176] synthesizer used similar decomposition rules to derive bitvector-manipulating programs. Using ideas from syntactically defined synthesis, some of the decomposition rules were non-deterministic, i.e., the rules were partial and were completed by the synthesizer into semantics-preserving rules.

Linear algebra The FLAME project uses linear algebra and Hoare-style proof rules to systematically derive an efficient algorithm for linear algebra operations such as triangular solvers [12,69]. The expert-guided derivation process starts from a worksheet, which syntactically defines the control-flow shape of the program to be derived, and proceeds by computing predicates that describe point-specific properties of matrix-valued variables. The program is then completed by adding executable statements that satisfy those predicates, viewing them as pre- and post-conditions. Since alternative predicates can be derived, the derivation process can produce alternative algorithms. This systematic process has been automated for at least one domain.

Polyhedral frameworks for matrix programs In the domain of programs that operate on dense matrices, polyhedral frameworks based on linear constraints over loop iteration spaces [47] have been used as foundations for algebras of loop transformations, such as loop reordering and loop blocking [6,22]. Strout et al. have extended dense polyhedral frameworks for sparse matrix codes [184]. The challenge in sparse matrix codes is in the indirect indexing of sparse matrices. Strout solves the problem by modeling the indirect index expression as a compile-time uninterpreted function; at runtime, the uninterpreted function is replaced with an inspector/executor algorithm that reorders loop iterations to satisfy all dependences of the sparse matrix.

Relational algebra Relational algebra has been used to describe container data structures in imperative programs [7,77,168]. Relational schemas describe data organization and relational queries model how a program uses the data structure. Efficient data structure are synthesized by producing more efficient schemas and modifying the original queries to work with the new schemas.

Statistical data analysis Fischer and Schumann [52] exploited an algebra based on probability theory and numerical theory to synthesize efficient statistical inference algorithms. Their system, AutoBayes, accepts a declarative description of random variables and uses schema-based synthesis to break down the declarative graphical model into a composition of algorithms. Symbolic reasoning based on the algebra is used to (i) test the applicability of algorithms expressed in the schema and (ii) obtain a closed-form solution when possible.

Arithmetic expression equivalence Optimizing compilers typically perform program optimizations using tree rewrite rules that capture equivalence of expressions, such as $x+0=x$, as well as equivalence of machine instructions. In compilers, these rewrites are typically applied heuristically. The Denali superoptimizer [89] produces all programs (up to a bound) derivable from the specification using a given set of expression equality axioms and selects the fastest program. The derivation exploits the E-graph data structure used in automatic theorem provers [40].

3.2 Syntactically defined synthesizers

Foundations for syntactically defined synthesizers have their origins in inductive inference of functions and formal languages from examples [5]. Inductive inference developed the crucial notion of *syntactic bias*, which is the restriction that the function to be induced must have a particular syntactic form. The central idea is that a syntactic restriction reduces the search space. A strong restriction, for example to a regular language, may additionally permit specialized algorithms.

The first well-known syntactic synthesizer of what one might call "programs" was SMARTedit, a programming-by-demonstration (PBD) system for synthesis of editing macros, by Lau et al. [112]. SMARTedit produced programs that were straight-line sequences of editing commands, for example, *move the cursor to end of line minus 5 positions; insert "hello"*. User demonstrations were sequences of the initial, intermediate, and final program states observed as the user manually carried out the editing sequence. The program state included the text buffer value, the cursor position, but not the command performed by the user, allowing the user some flexibility in achieving the editing action. The SMARTEdit designer provided the syntactic bias, i.e., the set of available commands and the syntactic form of their arguments, which were functions over the program state. The bias is expressed with version space algebra [114], in this setting equivalent to a subset of context-free grammars. The synthesis algorithm processed demonstrations one by one, maintaining the set of candidate programs consistent with demonstrations seen so far. The algorithm is particularly efficient because the program state in the demonstration included also the program point, which effectively decomposed the demonstrations into input–output specifications of individual commands, decomposing the synthesis problem into several smaller ones. The assumption that the demonstration includes program points was later lifted in [113], where programs with loops and conditionals are also synthesized.

The programming language aLisp (agent Lisp) by Andre et al. [4,130] was designed for expressing prior knowledge in hierarchical reinforcement learning of agent policies. A typical aLisp program would express the strategy of an agent playing a game such as Stratagus. The program would be non-deterministic in that certain low-level decisions would be left unspecified by the aLisp programmer. These decisions are predicates to be learnt (synthesized) by a reinforcement learning algorithm. The correctness specification was to maximize the reward function that measured the success of an agent in the actual game. Syntactic bias comes in two parts: (i) the aLisp *partial program* written by the agent programmer, who expresses his prior knowledge; and (ii) the syntax of the predicate to be learnt, fixed by the choice of the learning algorithm.

The Sketch language by Solar-Lezama et al. [177] introduces general-purpose partial programs, where programmers control the bias both via the partial programs and the syntax of code fragments that complete the "holes" in the partial programs. On the algorithmic side, Sketch showed how to solve the synthesis problem symbolically with a SAT solver, relying on the solver's learnt conflict clauses to prune the search of the candidate space. Sketch is described in this issue [173]. Template-based synthesis of program invariants [179] extended synthesis of invariants into synthesis of programs [180]. This work, by Srivastava et al., is also reported in this issue [181]. Itzhaky et al. [83] developed an efficient, tabular synthesis algorithm for partial programs that adhere to a special guarded-command form. Udupa et al. [188] showed that, in some cases, the synthesis process can proceed by exhaustive enumeration of the syntactically defined candidate space, rather than by symbolic means.

Genetic programming can also be viewed as syntactically defined synthesis [90,200]. Rather than defining the candidate space up front with a partial program, the candidate space in genetic programming is defined gradually as new candidate programs are obtained with mutations and other transformations of promising existing candidates.

Syntactic bias has also been used in controller synthesis. In [137], Morgenstern and Schneider considered the problem of expressing as partial programs a class of (non-terminating) reactive systems that should satisfy a formal specification given the full branching time logic CTL*. They showed how this partial programming problem can be reduced to a CTL* model checking problem. Madhusudan [122] used syntactically restricted tree automata.

4 Other approaches

There is a large variety of approaches and applications related to synthesis techniques that we would have liked to include in this journal but could not due to space restrictions. In this section we are trying to summary most of these.

Synthesis and theorem proving Theorem provers have been used for systematic semi-automated construction of programs from their logical specifications. The desired program or algorithm is specified in full first-order or second-order logic and the program is constructed from the proof of a theorem prover. Thanks to the high expressive power of these logics, this approach provides complete freedom as to what programs can be synthesized but the approach is based on human interaction in the program finding process. For example, the expert provides tactics and domain theories, e.g., the divide-and-conquer principle, to guide the underlying theorem prover (cf. the KIDS system [169] and Nuprl [11]).

Game theory Reactive synthesis is often based on automata-based game theory. Grädel et al. [62] edited a comprehensive guide to research in the areas of Automata, Logics, and Infinite Games.

Inductive programming Synthesis of programs from incomplete specifications presented as positive and negative examples [55] has been developed for functional programs [92] and logic programs [54,138].

Quantitative synthesis In several application areas of synthesis, researchers have observed the need for additional (non-functional) constraints, e.g., realizability and robustness. Such constraints are often expressed by assigning costs or rewards for certain actions of the controller. Reward and costs extensions have been considered for various types of systems (cf. [14,18,23,24,27,34–36,39,42,53]).

Timed and hybrid systems Maler et al. [124] presented algorithms for automatic synthesis of real-time controllers via solving timed-automata games [1]. Lygeros et al. [121] formulated the controller synthesis problems for reachability specifications in hybrid systems and presented a solution based on game theory. Synthesis for timed and hybrid systems is an active research area with new algorithms and tools developed in recent years (e.g., [9,41,117,132]).

Programming by demonstration The origins of programming by demonstration are in end-user-oriented systems [32]. Harel et al. developed Play-in/Play-out [76] based on Live Sequence Charts [75] for designers of reactive systems. Compositional synthesis from scenario-based reactive specification was done by Kugler et al. [101].

Requirement engineering Often, formalizing and debugging the specification is the most challenging aspect of system design. Requirements engineering synthesizes behavioral models (for e.g., for autonomous systems) from scenario-based requirements and visualizes declarative specifications for the purpose of their understanding [74,99,186,187,189, 201].

Distributed synthesis Synthesis of distributed controllers led to formulation of distributed games and synthesis problems with imperfect information [50,63,91,105,108,135,145].

5 Applications of synthesis

The synthesis techniques described in previous sections have been employed in a variety of applications. In this section, we organize related work by the application problem and highlight the key attributes of synthesis algorithms used for the problem.

Synthesis of client code Given a software library (e.g., a set of modules or classes), the goal is to compose library components into a client code with desirable functionality. The challenges of this problem include how to specify both the desired functionality and the behavior of existing components. The complex semantics of components requires modeling the behavior abstractly, for example with types. The consequence is that specifications are inherently ambiguous, necessitating programmer involvement in the synthesis process, which in turn requires research in new modes of programmer interaction and potentially incorporating secondary semantic information such as documentation expressed in natural language. An additional challenge is efficiency of synthesis algorithms for real-world object-oriented libraries, which may contain 10^5 to 10^6 methods.

The ETI system models component semantics with a type system and synthesizes the client code with a theorem prover [57,111,183]. Typsy [20] and Prospector [125] rely on the readily available Java static type annotations and address the ambiguity by ranking the candidate client code snippets. Strahcona [81] and ParseWeb [185] analyze code samples to mine candidate client code sequences. SNIFF [26] takes into consideration textual information in a code corpus. Reiss' semantic search [151] combines both static and dynamic information, while Matchmaker [203] collects extensive dynamic traces from which it builds a model library usage that is then used in synthesis. PROPHETS [140] synthesizes desired code based on types of components and SLTL constraints. Prime [134] mines temporal API specification, while InSynth Ruzica [71] develops a type system that prunes the search and ranks the solutions based on a code corpus.

Program and model repair and fault tolerance Shapiro [162] developed a general algorithm for bug localization that identifies inconsistencies in the program with respect to specifications of program's procedures. He also developed algorithmic repair for logic programs that operates via inductive inference [163].

Griesmayer et al. [64] developed a game-theoretical algorithm for repair of Boolean programs. In their formulation, if a memory-less winning strategy can be identified, then the Boolean program has been correctly repaired. Samanta et al. [155] showed how to repair Boolean programs by posing statement-level synthesis queries, which, if successful, will repair the program.

Demsky et al. [38] addressed bugs that cause data structure inconsistency by means of repairing not the program text but the value of its data structure, at runtime. Their algorithm searches a space of data structure edits to satisfy consistency specifications. Elkarablieh et al. [44] used static analysis to scale dynamic repair to large data structures.

Weimer et al. [200] formulated program repair via evolutionary programming. The space of candidate programs is defined by means of local program transformations which the evolutionary programming algorithm uses to derive programs that gradually pass more test cases. Samimi et al. [156] developed a solver-based algorithm for repair of PHP programs.

In [119], Logozzo and Ball studied the problem of suggesting code repairs at design time, based on the warnings issued by modular program verifiers.

In [196], von Essen and Jobstmann showed how to repair a reactive program with respect to a specification such that the repair does not introduce new bugs by requiring it to preserve correct behaviors.

Reactive synthesis techniques have been used to automatically add fault-tolerant behaviors to a given program (cf. [28,60,102]). Bonakdarpour and Kulkarni applied similar ideas in the context of revising distributed (UNITY) programs [21,43].

Synthesis of string manipulating functions Lau et al. [113] designed a PDB tool for synthesis of text editing macros, based on version space algebra and user demonstrations.

Gulwani [66] developed an algorithm for synthesis of spreadsheet text-manipulating macros, based on a carefully designed macro language of candidate macro programs and based on efficient algorithms for version space algebra. His algorithm has later appeared in Microsoft Office Excel under the name FlashFill. The algorithm was later extended to semantic manipulations based on domain theories of relations, currencies, dates, etc. [164].

Superoptimization The superoptimization problem is to find an optimal sequence of instructions that implements a certain function. The function is usually specified with a sub-optimal implementation.

The first superoptimizer was due to Massalin [131], whose algorithm enumerated all instruction sequences of up to size 4 or 5, testing them on a few inputs. This generate-and-test superoptimizer performed certain symmetry reductions, e.g., canonical register naming.

Joshi et al. [89] developed Denali, a superoptimizer that obtained the optimal sequence by rewriting the executable specification program into many alternative sequences using axioms similar to those employed in optimizing compilers. The resulting sequences were only as good as the sequence of axioms but they were correct by construction.

Jha et al. [67] developed a solver-based superoptimizer. Their algorithm also searches a space of all instruction sequences of up to a certain size but does so symbolically, that it can find instruction sequences of up to about 25 instructions. Correctness is verified symbolically, too.

Schkufza et al. [159] developed an algorithm that searches the candidate space stochastically, applying local mutations to program candidates that are deemed to be close to the correct program, based on a heuristic, test-based measure of correctness.

Mash-ups of web services and web scraping The Internet offers a diverse spectrum of web services, such as maps, dictionaries, and schedules of public transportation. These services are typically accessed from a web browser. When a user consults these services, he often manually composes them, e.g., when copying an address produced by a restaurant search into a mapping service. To avoid this manual composition, it is desirable to compose these services automatically, based on a user demonstration or some other specification.

Planning and synthesis techniques have been successfully used to compose web services, e.g., in [10,37].

Vegemite [116] produces browser scripts in the CoScripter language [115] based on user demonstrations.

Kubczak et al. [100] synthesized mash-ups using planning from service specifications produced in domain analysis.

A problem related to web service composition is that of extracting relational data from semistructured web pages. This problem is sometimes referred to as *web scraping*. Huynh et al. [82] developed an algorithm that synthesizes an extraction program from user demonstration. The demonstration identifies a sample or relational data; the extraction program is expressed as a path expression that identifies a relation embedded in the tree representation of the web page.

Data structure synthesis The problem of designing an efficient physical representation of a logical data structure can also be viewed as that of synthesis. Work in this area views the logical data structure as a relational store and synthesizes an efficient implementation starting from relational queries that characterize how the data structure is to be used by a client program [7,77,168]. Hawkins et al. [78] extended the work to the concurrent setting.

Database algorithms Spielmann et al. [95] showed how to automatically synthesize database algorithms that efficiently make use of memory hierarchy and external storage, starting from declarative descriptions that ignore memory access costs.

Cheung et al. [29] optimized code that uses a database by lifting, with synthesis, an imperative database client code fragment into a high-level relational query, which is then optimizable by an off-the-shelf database query planner.

Synthesis of efficient algorithms While most work in synthesis focuses on obtaining a relatively low-level *implementation*, some approaches could be considered to synthesize a high-level *algorithm*. Blaine et al. [13] synthesized highly scalable algorithms for logistical planning. Itzhaky et al. [83] synthesized incremental algorithms by allowing the user to provide a partial program (called a *target language*) that described a space of candidate incremental algorithms. Liu et al. developed an optimizing compiler that incrementalizes an object-oriented program. The technique, based on finite program differencing developed by Paige [142], comes with rewrite rules that introduce into objects auxiliary storage and simultaneously modify object methods to update this storage and exploit it for (asymptotically) more efficient optimizations. Pu et al. developed a synthesizer of linear-time dynamic programming algorithms based on partial programs.

Concurrent garbage collectors The challenge is to synthesize an algorithm that stops the mutator as little as possible while guaranteeing correctness and reducing garbage collection overhead. The Paraglide project developed techniques for synthesis of concurrent garbage collectors. Their CGCExplorer searches a space of candidate collectors that is constructed from building blocks selected by an expert according to his intuition as to how the collector should operate [193,194].

Compiling declarative specification Compiling declarative specifications into executable programs can also be considered to belong in program synthesis. Along these lines, Krishamurthi et al. [98] designed a compiler for alloy [84] specifications.

Aspect-oriented programming Maoz and Sa'ar [129] presented a framework based on reactive synthesis that allows the specification and implementation of crosscutting concerns using temporal logic.

Cache coherence protocols Cache coherence protocols implement a shared-memory programming model by governing how caches of a multiprocessor exchange cached memory locations. These protocols are distributed programs running at each cache as well as at the cache directory. Udupa et al. [188] synthesized the tricky guard expressions that control transitions among automata states that are maintained by the protocol.

Education Synthesis has been successfully applied in automating mundane as well as creative aspects of education. Singh et al. [165] extended the Sketch synthesizer to automatically grade Python programs. Gulwani et al. [68] designed algorithms that solve ruler-and-compass geometry problems. Seshia et al. [153] showed how to automatically generate problems in cyber-physical systems and Singh et al. [166] did so for algebra. Other systems that generate programs of controlled complexity are [182], in the controller domain, and [3], in algebra.

6 Summary of contributions

The contribution that perhaps best bridges the world of controller synthesis with the methods used for imperative programs is *Abstraction-Guided Synthesis of Synchronization*, by Vechev et al. [192]. Their goal is to synthesize synchronization for a multithreaded program so that a safety violation is avoided. In the framework of controller synthesis, this goal corresponds to synthesizing a controller that takes the form of atomic guards, over statements in the original (unsynchronized) program, which prevent the scheduler from preempting the running thread. From the algorithmic standpoint, the paper contributes a refinement algorithm whose novelty is that not only the abstraction, but also the program (i.e., the placement of atomic guards) is adjusted. The ability to modify the program allows authors to rule out program interleavings not only when they are known to be invalid, but also when they cannot be verified under the given abstraction.

Two papers exploit properties of specifications to design efficient synthesis algorithms. The first paper decomposes LTL specifications, while the second paper restricts itself to decidable logics to turn decision procedures into synthesis procedures. In *Safety First: A Two-Stage Algorithm for the Synthesis of Reactive Systems* [171], Sohail and Somenzi describe an algorithm that solves safety and persistence properties in the first step and accounts for liveness in the second step. This division allows a divide-and-conquer strategy in the first step, through merging of winning strategies for subproblems. The second step is a one-level divide-and-conquer algorithm because winning strategies for liveness need to be computed in a whole-program fashion. Liveness is computed with a symbolic conjunctive parity game, which manipulates characteristic functions rather than elements. In this paper, the specification is viewed as a conjunction of properties, assuming that each conjunct is small enough to allow explicit determinization. This paper may thus be of interest to the reader if her specification (i) contains a lot of small properties, (ii) uses the full expressive power of LTL (rather than the less expressive GR(1)), (iii) mostly contains safety or persistence properties, and (iv) contains the (general) liveness properties that are small enough to construct explicit deterministic parity automata. From the engineering standpoint, the paper offers a tunable tradeoff between the number of components of the conjunctive game and the size of individual properties, allowing one to conjoin these properties.

In *Functional Synthesis for Linear Arithmetic and Sets*, Kuncak et al. [104] integrate synthesis of functions into a host language, Scala, allowing one to program declaratively by expressing a relational specification over the inputs and outputs of the desired function. The function is synthesized conceptually by partial evaluation of a decision procedure with respect to the specification. To specialize a decision procedure, the authors observe that decision procedures are often based on quantifier elimination, which solves the constraint system by eliminating variables one at a time and then computing the variable values in reverse direction. It thus suffices to ask the quantifier elimination to produce a witness function, which computes the eliminated variable from the remaining variables. A suitable chaining of witness functions then constructs the synthesized function. The authors also help with debugging of declarative specifications. For example, when a specification is discovered to be underconstrained, the synthesizer identifies an input on which the function is allowed to return one of several values. Reporting the input to the programmer might hint at the constraint that is missing in the specification.

The next two papers synthesize imperative programs by reduction to constraint solving. Both papers allow the programmer to supply a partial program (called respectively a sketch or template), which is a program with holes that will be completed by the synthesizer. Partial programs both make the synthesis more efficient and allow the programmer to express his insight about the desired program.

In *Program Sketching*, Solar-Lezama [173] shows how to generate an implementation from a partial program using counterexample-guided inductive synthesis (CEGIS). Inductive synthesis is a process of generating candidate implementations from concrete examples of correct or incorrect behavior. CEGIS combines a SAT-based inductive synthesizer with an automated validation procedure, typically a bounded model, which supplies the concrete examples.

In *Template-based Program Verification and Program Synthesis*, Srivastava et al. [181] show how to complete a partial program by reducing the synthesis problem into a verification problem. The synthesizer is based on their prior work on template-based verifiers. While in the verifier it is the program invariant what is partially described with a template, in the synthesizer it is the program. Still, in an elegant twist, the synthesized program can be viewed as an invariant, providing the reduction.

The papers *Bounded Synthesis* by Finkbeiner and Schewe [51], and *Exploiting Structure in LTL Synthesis* by Filiot et al. [49] focus on synthesizing small reactive systems from arbitrary LTL specifications. They both search for all solutions that fall below a given bound on the size of the implementations. Incrementally increasing the bound allows them to obtain completeness for all decidable synthesis problems. Finkbeiner and Schewe were the first to present this idea in an initial version of their paper. In this paper, they also showed the generality of the approach by presenting experimental results indicating that many synthesis problems with generally intractable complexity can be solved efficiently for reasonably small bounds. Raskin, Jin, and Filiot observed that the resulting synthesis problems have a special structure that can be exploited for efficient implementations. Specifically, there is a partial order over the configurations used in the synthesis process. This partial order allows them to collapse configurations and represent them efficiently using a symbolic data structure based on anti-chains. In addition, they present a simplification of their original algorithm and a reduction to SAT solving which together lead to large gains in efficiency.

Synthesis is impossible without a correct specification. In *Debugging Formal Specifications*, Koenighofer et al. [96] develop algorithms for explaining unrealizable specifications to the specification author. Conceptually, this task boils down to pinpointing why none of the (infinitely many) implementations meets the specification. To sidestep enumerating incorrect implementations, the authors encouraged the user to find bugs in the informal design intent that exists only in his mind. The authors swapped the roles between the tool and the user: the tool plays the environment and the user plays the system to be synthesized, trying to meet the specification. The tool follows a counterstrategy that makes the user fail, forcing him to realize why the imagined implementation fails to meet the specification. This knowledge can then inform a fix to the specification.

The paper *Synthesis of AMBA AHB from Formal Specifications: A Case Study* by Godhal et al. [61] is intended for the reader interested in the state-of-the-art of automatically synthesizing integrated circuits from temporal specifications. The authors significantly improved the work of Bloem et al., who were the first to present a formal specification for ARM AMBA AHB Arbiter and synthesized the AHB Arbiter circuit. The Advanced Microcontroller Bus Architecture (AMBA) was developed by ARM, one of the leading companies in microprocessor Intellectual Property. AMBA is used as on-chip bus in a wide range of ASIC and SoC parts including processors used in smartphones. Chatterjee, Henzinger, and Godhal present detailed formal specifications for AHB Arbiter, and obtain significant improvement in synthesis results (both with respect to the number of gates in the synthesized circuit and with respect to time taken to synthesize the circuit). They also discuss principles for writing formal specifications for efficient hardware synthesis

Oddos et al. [141] also aim to synthesize a circuit from a set of formal properties. Their approach differed from reactive synthesis, as they turned each property into a component that combined classical monitor and generator features. The approach cannot handle arbitrary specifications but was extremely efficient, in particular, it allowed synthesizing circuits specified by hundred of temporal properties in a few seconds.

Finally, the paper *Synthesis from Component Libraries* by Lustig and Vardi [120] examines decidability questions of LTL program synthesis. Specifically, they examined component-based synthesis, an important direction for making synthesis more efficient through modularity. They introduced two notions of composition and described how to summarize a component so that synthesizer can compose it with the client of the component.

7 Conclusions

Despite extensively covering both fundamental techniques and practical applications of program synthesis, we were forced to leave out plenty of inspiring work. We trust that the interested reader will follow the citations from our bibliography to identify these papers. Readers interested in open problems that guide current research in synthesis will be served well by proceedings of conferences such as CAV, POPL, and PLDI, which typically include papers on the cutting edge of program synthesis.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)

2. Alur, R., La Torre, S.: Deterministic generators and games for LTL fragments. In: Symposium on Logic in Computer Science (LICS'01), pp. 291–302 (2001)
3. Andersen, E., Gulwani, S., Popovic, Z.: A trace-based framework for analyzing and synthesizing educational progressions. In: Mackay, W.E., Brewster, S.A., Bødker, S. (eds.), CHI. ACM, New York, pp. 773–782 (2013)
4. Andre, D., Russell, S.J.: State abstraction for programmable reinforcement learning agents. In: Dechter, R., Sutton, R.S. (eds.) AAAI/IAAI, pp. 119–125. AAAI Press/The MIT Press, Menlo Park (2002)
5. Angluin, D., Smith, C.H.: Inductive inference: theory and methods. ACM Comput. Surv. **15**(3), 237–269 (1983)
6. Bastoul, C., Cohen, A., Girbal, S., Sharma, S., Temam, O.: Putting polyhedral loop transformations to work. In: Languages and Compilers for Parallel Computing, pp. 209–225. Springer, Berlin (2004)
7. Batory, D.S., Thomas, J.: P2: a lightweight dbms generator. J. Intell. Inf. Syst. **9**(2), 107–123 (1997)
8. Baumgartner, J.: Integrating FV into main-stream verification: the IBM experience, 2006. Invited Talk at the Conference on Formal Methods in Computer Aided Design (FMCAD'06) (2006)
9. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: Uppaal-tiga: time for playing games! In: CAV, pp. 121–125 (2007)
10. Bertoli, P., Pistore, M., Traverso, P.: Automated composition of web services via planning in asynchronous domains. Artif. Intell. **174**(3), 316–361 (2010)
11. Bickford, M., Constable, R.L., Halpern, J.Y., Petride, S.: Knowledge-based synthesis of distributed systems using event structures. Log. Methods Comput. Sci., **7**(2:14), 1–36 (2011)
12. Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Ortí, E.S., van de Geijn, R.A.: The science of deriving dense linear algebra algorithms. ACM Trans. Math. Softw. **31**(1), 1–26 (2005)
13. Blaine, L., Gilham, L., Liu, J., Smith, D.R., Westfold, S.J.: Planware—domain-specific synthesis of high-performance schedulers. In: ASE, p. 270 (1998)
14. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.), CAV. Lecture Notes in Computer Science, Vol. 5643, pp. 140–156. Springer, Berlin (2009)
15. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Koenighofer, R., Roveri, M., Schuppan, V., Seeber, R.: Ratsy—a new requirements analysis tool with synthesis. In: Comput. Aided Verification (2010, To appear)
16. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic hardware synthesis from specifications: a case study. In: DATE (2007)
17. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Martin W.: Specify, compile, run: hardware from PSL. In: COCV, Electronic Notes in Computer Science, pp. 3–16 (2007)
18. Bloem, R., Greimel, K., Henzinger, T.A., Jobstmann, B.: Synthesizing robust systems. In: FMCAD, pp. 85–92 (2009)
19. Boehm, H.-J., Flanagan, C.: (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, 16–19 June 2013. ACM, New York (2013)
20. Bolton, C., Nelson, G.: Typsy: a type-based search tool for java programmers. Technical Report SRC Technical Note, 2001–004 (Selected: SRC Summer Intern Reports). Compaq SRC, December (2001)
21. Bonakdarpour, B., Kulkarni, S.S.: Sycraft: a tool for synthesizing distributed fault-tolerant programs. In: Breugel, F., Chechik, M. (eds.), CONCUR 2008—Concurrency Theory. Lecture Notes in Computer Science, vol. 5201, pp. 167–171. Springer, Berlin (2008)
22. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Gupta, R., Amarasinghe, S.P. (eds.), PLDI, pp. 101–113. ACM, New York (2008)
23. Bouyer, P.: Weighted timed automata: model-checking and games. In: Brookes, S., Mislove, M. (eds.), Proceedings of the 22nd Conference on Mathematical Foundations of Programming Semantics (MFPS'06). Electronic Notes in Theoretical Computer Science, vol. 158, pp. 3–17, Genova, Italy, May 2006. Elsevier Science Publishers, Amsterdam (2006, Invited paper)
24. Bouyer, P., Markey, N., Sankur, O.: Robust weighted timed automata and games. In: Braberman, V., Fribourg, L. (eds.), Proceedings of the 11th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'13). Lecture Notes in Computer Science, Buenos Aires, Argentina, August 2013. Springer, Berlin (2013, To appear)
25. Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Trans. Am. Math. Soc. **138**, 295–311 (1969)
26. Chatterjee, S., Juvekar, S., Sen, K.: Sniff: a search engine for java using free-form queries. In: Chechik, M., Wirsing, M. (eds.), FASE. Lecture Notes in Computer Science, vol. 5503, pp. 385–400. Springer, Berlin (2009)
27. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: a model checker for stochastic multi-player games. In: Piterman, N., Smolka, S. (eds.), Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13). LNCS, vol. 7795, pp. 185–191. Springer, Berlin (2013)
28. Cheng, C.-H., Rueß, H., Knoll, A., Buckl, C.: Synthesis of fault-tolerant embedded systems using games: from theory to practice. In: Verification, Model Checking, and Abstract Interpretation, pp. 118–133. Springer, Berlin (2011)
29. Cheung, A., Solar-Lezama, A., Madden, S.: Optimizing database-backed applications with query synthesis. In: Boehm and Flanagan, vol. 19, pp. 3–14
30. Church, A.: Logic, arithmetic and automata. In: Proceedings International Mathematical Congress (1962)
31. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
32. Cypher, A., Dontcheva, M., Lau, T., Nichols, J.: No Code Required: Giving Users Tools to Transform the Web. Morgan Kaufmann Publishers Inc., San Francisco (2010)
33. Daniele, M., Giunchiglia, F., Vardi, M.Y.: Improved automata generation for linear time temporal logic. In: Halbwachs, N., Peled, D. (eds.), Eleventh Conference on Computer Aided Verification (CAV'99), LNCS 1633, pp. 249–260. Springer, Berlin (1999)
34. de Luca, A.: How to specify and verify the long-run average behavior of probabilistic systems. In: LICS, pp. 454–465 (1998)
35. de Luca, A., Henzinger, T.A., Majumdar, R.: Discounting the future in systems theory. In: ICALP, pp. 1022–1037 (2003)
36. de Luca, A., Majumdar, R., Raman, V., Stoelinga, M.: Game relations and metrics. In: LICS, pp. 99–108 (2007)
37. De Giuseppe, G., Patrizi, F.: Automated composition of nondeterministic stateful services. In: Web Services and Formal Methods, pp. 147–160. Springer, Berlin (2010)
38. Demsky, B., Rinard, M.C.: Goal-directed reasoning for specification-based data structure repair. IEEE Trans. Softw. Eng. **32**(12), 931–951 (2006)
39. Desharnais, J., Gupta, V., Jagadeesan, R.: Metrics for labelled markov processes. Theor. Comput. Sci. **318**(3), 323–354 (2004)
40. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
41. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: CAV, pp. 167–170 (2010)

42. Doyen, L., Henzinger, T.A., Legay, A., Nickovic, D.: Robustness of sequential circuits. In: ACSD, pp. 77–84 (2010)
43. Ebnenasir, A., Kulkarni, S.S., Bonakdarpour, B.: Revising unity programs: possibilities and limitations. In: Principles of Distributed Systems, pp. 275–290. Springer, Berlin (2006)
44. Elkarablieh, B., Khurshid, S., Vu, D., McKinley, K.S.: Starc: static analysis for efficient repair of complex data. In: ACM SIGPLAN Notices, vol. 42, pp. 387–404. ACM, New York (2007)
45. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. **2**, 241–266 (1982)
46. Etessami, K., Holzmann, G.J.: Optimizing Büchi automata. In: Proceedings of the 11th International Conference on Concurrency Theory (CONCUR2000), LNCS 1877, pp. 153–167. Springer, Berlin (2000)
47. Feautrier, P.: Automatic parallelization in the polytope model. In: Perrin, G., Darte, A. (eds.), The Data Parallel Programming Model. Lecture Notes in Computer Science, vol. 1132, pp. 79–103. Springer, Berlin (1996)
48. Filiot, E., Jin, N., Raskin, J.-F.: An antichain algorithm for ltl realizability. In: Proceedings of the Computer Aided Verification, pp. 263–277 (2009)
49. Filiot, Emmanuel: Jin, N., Raskin. J.-F.. Exploiting structure in ltl synthesis, STTT (2013, in this issue)
50. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, 2005. LICS 2005, pp. 321–330. IEEE, New York (2005)
51. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT (2013, in this issue)
52. Fischer, B., Schumann, J.: Autobayes: a system for generating data analysis programs from statistical models. J. Funct. Program. **13**(3), 483–508 (2003)
53. Fisman, D., Kupferman, O., Lustig, Y.: Rational synthesis. In: TACAS, pp. 190–204 (2010)
54. Flener, P.: Logic program synthesis from incomplete information. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston (1995)
55. Flener, P., Schmid, U.: An introduction to inductive programming. Artif. Intell. Rev. **29**(1), 45–62 (2008)
56. Franchetti, F., de Mesmay, F., McFarlin, D.S., Püschel, M.: Operator language: a program generation framework for fast kernels. In: Taha, W.M. (ed.), DSL. Lecture Notes in Computer Science, vol. 5658, pp. 385–409. Springer, Berlin (2009)
57. Freitag, B., Margaria, T., Steffen, B.: A pragmatic approach to software synthesis. SIGPLAN Not. **29**(8), 46–58 (1994)
58. Frigo, M.: A fast fourier transform compiler. In: Ryder, B.G., Zorn, B.G. (eds.), PLDI, pp. 169–180. ACM, New York (1999)
59. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.), Thirteenth Conference on Computer Aided Verification (CAV '01), LNCS 2102, pp. 53–65. Springer, Berlin (2001)
60. Girault, A., Rutten, E.: Automating the addition of fault tolerance with discrete controller synthesis. Formal Methods in System Design **35**(2), 190–225 (2009)
61. Godhal, Y., Chatterjee, K., Henzinger, T.A.: Synthesis of amba ahb from formal specifications: a case study. STTT (2013, in this issue)
62. Grädel, E., Thomas, W., Wilke, T. (eds.) Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]. Lecture Notes in Computer Science, vol. 2500. Springer, Berlin (2002)
63. Graf, S., Peled, D., Quinton, S.: Achieving distributed control through model checking. Formal Methods Syst. Des. **40**(2), 263–281 (2012)
64. Griesmayer, A., Bloem, R., Cook, B.: Repair of boolean programs with an application to c. In: Ball, T., Jones, R.B. (eds.), CAV. Lecture Notes in Computer Science, vol. 4144. Springer, Berlin (2006)
65. Gulwani, S.: Dimensions in program synthesis. In: Bloem, R., Sharygina, N. (eds.), FMCAD, p. 1. IEEE, New York (2010)
66. Gulwani, S.: Automating string processing in spreadsheets using input–output examples. In: Ball, T., Sagiv, M. (eds.), POPL, pp. 317–330. ACM, New York (2011)
67. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Hall, M.W., Padua, D.A. (eds.), Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 62–73. ACM, New York (2011)
68. Gulwani, S., Korthikanti, V.A., Tiwari, A.: Synthesizing geometry constructions. In: Hall, M.W., Padua, D.A. (eds.), Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 50–61. ACM, New York (2011)
69. Gunnels, J.A., Gustavson, F.G., Henry, G., van de Geijn, R.A.: Flame: formal linear algebra methods environment. ACM Trans. Math. Softw. **27**(4), 422–455 (2001)
70. Gurevich, Y., Harrington, L.: Trees, automata, and games. In: Proceedings of the 14th ACM Symposium. Theory of Comp., pp. 60–65, San Francisco (1982)
71. Gvero, T.: Kuncak, V., Kuraj, I., Piskac, R.: Complete completion using types and weights. In: ACM SIGPLAN PLDI, Ruzica (2013)
72. Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, 4–8 June 2011. ACM, New York (2011)
73. Harding, A., Ryan, M., Schobbens, P.-Y.: A new algorithm for strategy synthesis in LTL games. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), pp. 477–492, Edinburgh. LNCS 3440 (2005)
74. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: generating statechart models from scenario-based requirements. In: Formal Methods in Software and Systems Modeling, pp. 309–324. Springer, Berlin (2005)
75. Harel, David, Marelly, Rami: Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine. Springer, Secaucus (2003)
76. Harel, D., Segall, I.: Synthesis from scenario-based specifications. J. Comput. Syst. Sci. **78**(3), 970–980 (2012)
77. Hawkins, P., Aiken, A., Fisher, K., Rinard, M.C., Sagiv, M.: Data representation synthesis. In: Hall and Padua, vol. 72, pp. 38–49
78. Hawkins, P., Aiken, A., Fisher, K., Rinard, M.C., Sagiv, M.: Concurrent data representation synthesis. In: Vitek, J., Lin, H., Frank, T. (eds.) PLDI, pp. 417–428. ACM, New York (2012)
79. Henzinger, T.A., Piterman, N.: Solving games without determinization. In: Proceedings of the 15th Conference on Computer Science Logic, pp. 395–410 (2006)
80. Kurshan, R.P., Touati, H.J., Brayton, R.K.: Testing language containment for ω-automata using BDD's. Inf. Comput. **118**(1), 101–109 (1995)
81. Holmes, R., Walker, R.J., Murphy, G.C.: Strathcona example recommendation tool. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pp. 237–240. ACM, New York (2005)
82. Huynh, D.F., Miller, R.C., Karger, D.R.: Enabling web browsers to augment web sites' filtering and sorting functionalities. In: Proceedings of the 19th Annual ACM Symposium on User Interface

Software and Technology, UIST '06, pp. 125–134. ACM, New York (2006)

83. Itzhaky, S., Gulwani, S., Immerman, N., Sagiv, M.: A simple inductive synthesis methodology and its applications. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.), OOPSLA, pp. 36–46. ACM, New York (2010)
84. Jackson, D.: Alloy: a new technology for software modelling. In: Katoen, J.-P., Stevens, P. (eds.), TACAS. Lecture Notes in Computer Science, vol. 2280, p. 20. Springer, Berlin (2002)
85. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE 2010 (2010)
86. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: Conference on Formal Methods in Computer Aided Design, pp. 117–124 (2006)
87. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: a tool for property synthesis. In: Comput. Aided Verif., pp. 258–262 (2007)
88. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.), 17th Conference on Computer Aided Verification (CAV'05), pp. 226–238. Springer, Berlin. LNCS 3576 (2005)
89. Joshi, R., Nelson, G., Zhou, Y.: Denali: a practical algorithm for generating optimal code. ACM Trans. Program. Lang. Syst. **28**(6), 967–989 (2006)
90. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: Ramakrishnan, C.R., Rehof, J. (eds.), TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 141–156. Springer, Berlin (2008)
91. Katz, G., Peled, D., Schewe, S.: Synthesis of distributed control through knowledge accumulation. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. Lecture Notes in Computer Science, vol. 6806. Springer, Brelin (2011)
92. Kitzelmann, E.: Inductive programming: a survey of program synthesis techniques. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) AAIP. Lecture Notes in Computer Science, vol. 5812, pp. 50–73. Springer, Berlin (2009)
93. Klarlund, N.: Progress measures for complementation of ω-automata with application to temporal logic. In: Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science, pp. 358–367, San Juan (1991)
94. Klein, Joachim, Baier, Christel.: Experiments with deterministic omega-automata for formulas of linear temporal logic. In: Conference on Implementation and Application of Automata (CIAA'05), pp. 199–212. LNCS 3845 (2005)
95. Klonatos, Y., Nötzli, A., Spielmann, A., Koch, C., Kuncak, V.: Automatic synthesis of out-of-core algorithms. In: Ross, K.A., Srivastava, D., Papadias, D. (eds.) SIGMOD Conference, pp. 133–144. ACM, New York (2013)
96. Könighofer, R., Hofferek, G., Bloem, R.: Debugging formal specications—a practical approach using model-based diagnosis and counterstrategies. STTT (2013, in this issue)
97. Kretínský, J., Esparza, J.: Deterministic automata for the (f, g)-fragment of ltl. In: CAV, pp. 7–22 (2012)
98. Krishnamurthi, S., Fisler, K., Dougherty, D.J., Yoo, D.: Alchemy: transmuting base alloy specifications into implementations. In: Harrold, M.J., Murphy, G.C. (eds.) SIGSOFT FSE. ACM, New York (2008)
99. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From mscs to statecharts. In: Proceedings of the IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems, DIPES '98, pp. 61–71, Norwell. Kluwer Academic Publishers, Dordrecht (1999)
100. Kubczak, C., Margaria, T., Steffen, B.: Mashup development for everybody: a planning—based approach. In: Proceedings of the 3rd International Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web, SMR2 (2009)
101. Kugler, H., Segall, I.: Compositional synthesis of reactive systems from live sequence chart specifications. In: TACAS, pp. 77–91 (2009)
102. Kulkarni, S., Arora, A.: Automating the addition of fault-tolerance. In: Joseph, M. (ed.), Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 1926, pp. 82–93. Springer, Berlin (2000)
103. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI (2010)
104. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT (2013, in this issue)
105. Kupermann, O., Varfi, M.Y.: Synthesizing distributed systems. In: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, 2001, pp. 389–398. IEEE, New York (2001)
106. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Eighteenth Conference on Computer Aided Verification, pp. 31–44, LNCS 4144 (2006)
107. Kupferman, O., Vardi, M.Y.: Freedom, weakness, and determinism: from linear-time to branching-time. In: Proceedings of the 13th IEEE Symposium on Logic in Computer Science (1998)
108. Kupferman, O., Vardi, M.Y.: Synthesis with incomplete information. Adv. Temp. Logic **16**, 109–127 (2000)
109. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Foundations of Computer Science, pp. 531–542, Pittsburgh, (2005)
110. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000)
111. Lamprecht, A.-L., Margaria, T., Steffen, B.: Bio-jeti: a framework for semantics-based service composition. BMC Bioinf. **10**(S–10), 8 (2009)
112. Lau, T.A., Domingos, P., Weld, D.S.: Version space algebra and its application to programming by demonstration. In: Langley, P. (ed.), ICML, pp. 527–534. Morgan Kaufmann, Burlington (2000)
113. Lau, T.A., Domingos, P., Weld, D.S.: Learning programs from traces using version space algebra. In: Gennari, J.H., Porter, B.W., Gil, Y. (eds.), K-CAP, pp. 36–43. ACM, New York (2003)
114. Lau, T.A., Wolfman, S.A., Domingos, P., Weld, D.S.: Programming by demonstration using version space algebra. Machine Learn. **53**(1–2), 111–156 (2003)
115. Leshed, G., Haber, E.M., Matthews, T., Lau, T.A.: Coscripter: automating & sharing how-to knowledge in the enterprise. In: Czerwinski, M., Lund, A.M., Tan, D.S. (eds.), CHI, pp. 1719–1728. ACM, New York (2008)
116. Lin, J., Wong, J., Nichols, J., Cypher, A., Lau, T.A.: End-user programming of mashups with vegemite. In: Conati, C., Bauer, M., Oliver, N., Weld, D.S. (eds.), IUI, pp. 97–106. ACM, New York (2009)
117. Liu, J., Ozay, N., Topcu, U., Murray, R.M.: Switching protocol synthesis for temporal logic specifications. In: American Control Conference (ACC), 2012, pp. 727–734. IEEE, New York (2012)
118. Löding, C.: Optimal bounds for transformations of omega-automata. In: Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 97–109 (1999)
119. Logozzo, F., Ball, T.: Modular and verified automatic program repair. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp. 133–146. ACM, New York (2012)
120. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. STTT (2013, in this issue)
121. Lygeros, J., Tomlin, C., Sastry, S.: Controllers for reachability specifications for hybrid systems. Automatica **35**(3), 349–370 (1999)

122. Madhusudan, P.: Synthesizing reactive programs. Proceedings of the Comp. Sci. Log., CSL 2011, pp. 428–442 (2011)
123. Maidl, M.: The common fragment of CTL and LTL. In: Proceedings of the 41th Annual Symposium on Foundations of Computer Science, pp. 643–652 (2000)
124. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (an extended abstract). In: STACS, pp. 229–242 (1995)
125. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the api jungle. In: Sarkar, V., Hall, M.W. (eds.), PLDI, pp. 48–61. ACM, New York (2005)
126. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems *Specification*. Springer, Berlin (1991)
127. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, Berlin (1995)
128. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Trans. Program. Lang. Syst. **6**, 68–93 (1984)
129. Maoz, S., Sa'ar, Y.: Aspectltl: an aspect language for ltl specifications. In: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development, pp. 19–30. ACM, New York (2011)
130. Marthi, B., Russell, S.J., Latham, D., Guestrin, C.: Concurrent hierarchical reinforcement learning. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI. Professional Book Center, Mumbai (2005)
131. Massalin, H.: Superoptimizer—a look at the smallest program. In: Katz, R.H. (ed.) ASPLOS, pp. 122–126. ACM Press, New York (1987)
132. Mazo Jr, M., Davitian, A., Tabuada, P.: Pessoa: a tool for embedded controller synthesis. In: Computer Aided Verification, pp. 566–569. Springer, Berlin (2010)
133. Michel, M.: Complementation is more difficult with automata on infinite words. Manuscript, CNET, Paris (1988)
134. Mishne, A., Shoham, S., Yahav, E.: Typestate-based semantic code search over partial programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, pp. 997–1016. ACM, New York (2012)
135. Mohalik, S., Walukiewicz, I.: Distributed games. In: FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, pp. 338–351. Springer, Berlin (2003)
136. Morgenstern, A.: Symbolic Controller Synthesis for LTL Specifications. PhD thesis, Universität Kaiserslautern (2010)
137. Morgenstern, A., Schneider, K.: Program sketching via ctl* model checking. In: Model Checking Software, pp. 126–143. Springer, Berlin (2011)
138. Muggleton, S.: Inductive logic programming. New Gener. Comput. **8**(4), 295–318 (1991)
139. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of Rabin, McNaughton and Safra. Theor. Comput. Sci. **141**, 69–107 (1995)
140. Naujokat, S., Lamprecht, A.-L., Steffen, B.: Loose programming with PROPHETS. In: de Lara, J., Zisman, A. (eds.), FASE. Lecture Notes in Computer Science, vol. 7212, pp. 94–98. Springer, Berlin (2012)
141. Oddos, Y., Morin-Allory, K., Borrione, D.: From assertion-based verification to assertion-based synthesis. In: VLSI-SoC: Technologies for Systems Integration, pp. 94–117. Springer, Berlin (2011)
142. Paige, R.: Symbolic finite differencing—part i. In: Jones, N.D. (ed.), ESOP. Lecture Notes in Computer Science, vol. 432, pp. 36–56. Springer, Berlin (1990)
143. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: 21st Symposium on Logic in Computer Science, pp. 255–264, Seattle (2006)
144. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: 7th International Conference on Verification, Model Checking and Abstract Interpretation, pp. 364–380. Springer, Berlin. LNCS 3855 (2006)
145. Pneuli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: 31st Annual Symposium on Foundations of Computer Science, 1990. Proceedings, pp. 746–757. IEEE, New York (1990)
146. Pnueli, A.: The temporal logic of programs. In: IEEE Symposium on Foundations of Computer Science, pp. 46–57, Providence (1977)
147. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the Symposium on Principles of Programming Languages (POPL '89), pp. 179–190 (1989)
148. Pueschel, M., Franchetti, F., Voronenko, Y.: Encyclopedia of Parallel Computing. In: Padua, D.A. (ed.), chapter Spiral. Springer Reference, Berlin (2011)
149. Rabin, M.O.: Automata on Infinite Objects and Church's Problem. Regional Conference Series in Mathematics. American Mathematical Society, Providence (1972)
150. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proc. IEEE **77**, 81–98 (1989)
151. Reiss, S.P.: Semantics-based code search. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 243–253, Washington, DC. IEEE Computer Society, New York (2009)
152. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, Weizmann Institute of Science (1992)
153. Sadigh, D., Seshia, S.A., Gupta, M.: Automating exercise generation: a step towards meeting the MOOC challenge for embedded systems. In: Proceedings of the Workshop on Embedded Systems Education (WESE) (2012)
154. Safra, S.: On the complexity of ω-automata. In: Symposium on Foundations of Computer Science, pp. 319–327 (1988)
155. Samanta, R., Deshmukh, J.V., Emerson, E.A.: Automatic generation of local repairs for boolean programs. In: Formal Methods in Computer-Aided Design, 2008. FMCAD'08, pp. 1–10. IEEE, New York (2008)
156. Samimi, H., Schäfer, M., Artzi, S., Millstein, T.D., Tip, F., Hendren, L.J.: Automated repair of html generation errors in php applications using string constraint solving. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.), ICSE, pp. 277–287. IEEE, New York (2012)
157. Sandryhaila, A., Kovacevic, J., Püschel, M.: Algebraic signal processing theory: cooley-tukey-type algorithms for polynomial transforms based on induction. SIAM J. Matrix Anal. Appl. **32**(2), 364–384 (2011)
158. Schewe, S.: Bounded synthesis. In: Automated Technology for Verification and Analysis, pp. 474–488 (2007)
159. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Sarkar, V., Bodík, R. (eds.) ASPLOS. ACM, New York (2013)
160. Althoff, C.S., Thomas, W., Wallmeier, N.: Observations on determinization of Büchi automata. Theor. Comput. Sci. **363**, 224–233 (2006)
161. Seshia, S.A.: Sciduction: combining induction, deduction, and structure for and synthesis. In: Groeneveld, P., Sciuto, D., Hassoun, S. (eds.) DAC. ACM, New York (2012)
162. Shapiro, E.Y.: Algorithmic program diagnosis. In: DeMillo, R.A. (ed.) POPL, pp. 299–308. ACM Press, New York (1982)
163. Shapiro, E.Y.: Algorithmic Program DeBugging. MIT Press, Cambridge (1983)

164. Singh, R., Gulwani, S.: Learning semantic string transformations from examples. PVLDB **5**(8), 740–751 (2012)
165. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: Boehm, H.-J., Flanagan, C. (eds.), Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'13, pp. 15–26. ACM, New York (2013)
166. Singh, R., Gulwani, S., Rajamani, S.K.: Automatically generating algebra problems. In: Hoffmann, J., Selman, B. (eds.) AAAI. AAAI Press, Menlo Park (2012)
167. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logic. J. ACM **3**(32), 733–749 (1985)
168. Smaragdakis, Y., Batory, D.S.: Distil: a transformation library for data structures. In DSL. USENIX (1997)
169. Smith, D.R.: Kids: a semiautomatic program development system. IEEE Trans. Softw. Eng. **16**(9), 1024–1043 (1990)
170. Sohail, S., Somenzi, F.: Safety first: a two-stage algorithm for LTL games. In: FMCAD'09, pp. 77–84. IEEE Press, New York (2009)
171. Sohail, S., Somenzi, F.: Safety first: a two-stage algorithm for the synthesis of reactive systems. STTT (2013, in this issue)
172. Sohail, S., Somenzi, F., Ravi, K.: A hybrid algorithm for LTL games. In: VMCAI. LNCS, vol. 4905, pp. 309–323. Springer, Berlin (2008)
173. Solar-Lezama, A.: Program sketching. STTT (2013, in this issue)
174. Solar-Lezama, A., Arnold, G., Tancau, L., Bodík, R., Saraswat, V.A., Seshia, S.A.: Sketching stencils. In: PLDI, pp. 167–178 (2007)
175. Solar-Lezama, A., Jones, C.G., Bodík, R.: Sketching concurrent data structures. In: PLDI, pp. 136–148 (2008)
176. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: PLDI, pp. 281–294 (2005)
177. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415 (2006)
178. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Emerson, E.A., Sistla, A.P. (eds.), Twelfth Conference on Computer Aided Verification (CAV'00), pp. 248–263. Springer, Berlin. LNCS 1855 (2000)
179. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: Hind, M., Diwan, A. (eds.), PLDI, pp. 223–234. ACM, New York (2009)
180. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL, pp. 313–326 (2010)
181. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verication and program synthesis. STTT (2013, in this issue)
182. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation. In: Bartocci, E., Ramakrishnan, C.R. (eds.), SPIN. Lecture Notes in Computer Science, vol. 7976, pp. 341–357. Springer, Berlin (2013)
183. Steffen, B., Margaria, T., Braun, V.: The electronic tool integration platform: concepts and design. Int. J. Softw. Tools Technol. Transfer **1**(1–2), 9–30 (1997)
184. Strout, M.M., Georg, G., Olschanowsky, C.: Set and relation manipulation for the sparse polyhedral framework. In: Kasahara, H., Kimura, K. (eds.), LCPC. Lecture Notes in Computer Science, vol. 7760, pp. 61–75. Springer, Berlin (2012)
185. Thummalapenta, S., Xie, T.: Parseweb: a programmer assistant for reusing open source code on the web. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) ASE. ACM, New York (2007)
186. Uchitel, S., Brunet, G., Chechik, M.: Behaviour model synthesis from properties and scenarios. In: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, pp. 34–43, Washington, DC. IEEE Computer Society, New York (2007)
187. Uchitel, S., Kramer, J., Magee, J.: Synthesis of behavioral models from scenarios. IEEE Trans. Softw. Eng. **29**(2), 99–115 (2003)
188. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: Transit: specifying protocols with concolic snippets. In: Boehm and Flanagan, vol. 19, pp. 287–296
189. Van, H.T., van Lamsweerde, A., Massonet, P., Ponsard, C.: Goal-oriented requirements animation. In: Proceedings of the Requirements Engineering Conference, 12th IEEE International, RE '04, pp. 218–228, Washington, DC. IEEE Computer Society, New York (2004)
190. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. Society for Industrial and Applied Mathematics, Philadelphia (1992)
191. Vardi, M.Y.: Branching vs. linear time: final showdown. Lecture Notes in Computer Science **2031**, 1–22 (2001)
192. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. STTT (2013, in this issue)
193. Vechev, M.T., Yahav, E., Bacon, D.F.: Correctness-preserving derivation of concurrent garbage collection algorithms. In: Schwartzbach, M.I., Ball, T. (eds.) PLDI. ACM, New York (2006)
194. Vechev, M.T., Yahav, E., Bacon, D.F., Rinetzky, N.: Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors. In: Ferrante, J., McKinley, K.S. (eds.) PLDI. ACM, New York (2007)
195. Vechev, M.T., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 5505, pp. 139–154. Springer, Berlin (2009)
196. von Essen, C., Jobstmann, B.: Program repair without regret. In: CAV 2013. Springer, Berlin (2013)
197. Wallmeier, N., Hütten, P., Thomas, W.: Symbolic synthesis of finite-state controllers for request–response specifications. In: Proceedings of the International Conference on the Implementation and Application of Automata. Springer, Berlin (2003)
198. Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., Mahlke, S.A.: Gadara: dynamic deadlock avoidance for multithreaded programs. In: Draves, R., van Renesse, R. (eds.) OSDI, pp. 281–294. USENIX Association, Berkeley (2008)
199. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.A.: The theory of deadlock avoidance via discrete control. In: POPL, pp. 252–263 (2009)
200. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Commun. ACM **53**(5), 109–116 (2010)
201. Whittle, J., Jayaraman, P.K.: Generating hierarchical state machines from use case charts. In: Proceedings of the 14th IEEE International Requirements Engineering Conference, RE '06, pp. 16–25, Washington, DC. IEEE Computer Society, New York (2006)
202. Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths. In: Proceedings of the 24th IEEE Symposium on Foundations of Computer Science, pp. 185–194 (1983)
203. Yessenov, K., Xu, Z., Solar-Lezama, A.: Data-driven synthesis for object-oriented frameworks. In: Lopes, C.V., Fisher, K. (eds.) OOPSLA. ACM, New York (2011)