From Boolean to Quantitative Synthesis *

Pavol Černý
IST Austria
cernyp@ist.ac.at

Thomas A. Henzinger
IST Austria
tah@ist.ac.at

ABSTRACT

Motivated by improvements in constraint-solving technology and by the increase of routinely available computational power, *partial-program synthesis* is emerging as an effective approach for increasing programmer productivity. The goal of the approach is to allow the programmer to specify a part of her intent imperatively (that is, give a partial program) and a part of her intent declaratively, by specifying which conditions need to be achieved or maintained. The task of the synthesizer is to construct a program that satisfies the specification. As an example, consider a partial program where threads access shared data without using any synchronization mechanism, and a declarative specification that excludes data races and deadlocks. The task of the synthesizer is then to place locks into the program code in order for the program to meet the specification.

In this paper, we argue that *quantitative objectives* are needed in partial-program synthesis in order to produce higher-quality programs, while enabling simpler specifications. Returning to the example, the synthesizer could construct a naive solution that uses one global lock for shared data. This can be prevented either by constraining the solution space further (which is error-prone and partly defeats the point of synthesis), or by optimizing a quantitative objective that models performance. Other quantitative notions useful in synthesis include fault tolerance, robustness, resource (memory, power) consumption, and information flow.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*computer-aided software engineering*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

*This work was partially supported by the ERC Advanced Grant QUAREM, the FWF NFN Grant S11402-N23 (RiSE), and the EU NOE Grant ArtistDesign.

EMSOFT'11, October 9–14, 2011, Taipei, Taiwan.

General Terms

Verification, Algorithms

Keywords

synthesis, quantitative models, quantitative synthesis

1. INTRODUCTION

While the automatic synthesis of programs has long been studied from a theoretical point of view, it has recently attracted significant research interest as a way to increase programmer productivity. The problem of synthesis from specifications was originally posed by Church [16]. Manna and Waldinger [31] and Biermann [1] considered the deductive synthesis of functions from relational specifications. Their approach is based on the idea that a program can be synthesized from the proof of the specification. The synthesis system KIDS [35] by Smith uses a similar approach. Clarke and Emerson [18] presented an algorithm for the synthesis of synchronization skeletons. The synthesis of reactive systems from temporal logic specifications was considered in [33] by Pnueli and Rosner. More recently, program synthesis has received renewed research attention mainly due to work by Solar-Lezama, Bodík, and others [38, 36, 37], who showed how program synthesis can be useful in practice. Their approach was originally called *sketching*, and by several other terms subsequently. We will use the descriptive name *partial-program synthesis*.

The goal of partial-program synthesis is "less ambitious" than the goal of classical synthesis. Rather than to automatically construct a program from its specification, the aim is to automate a completion of a partially written program. For instance, if the partial program already defines the intended functionality, only nonfunctional aspects (such as synchronization, security, fault tolerance, and resource use) need to be synthesized.

Boolean synthesis corresponds to finding one of the many possible solutions of a given constraint satisfaction problem. Quantitative synthesis, on the other hand, corresponds to finding an optimal solution. Specifying optimality requires the addition of an objective function to the specification. The objective function may refer, for example, to the use of resources (time, memory, power), or to the level of robustness or fault tolerance.

2. BOOLEAN PARTIAL-PROGRAM SYNTHESIS

The goal of partial-program synthesis is to allow the programmer to specify a part of her intent imperatively as a partial program, and a part of her intent declaratively, by specifying conditions that need to be achieved or maintained. The synthesizer then constructs a program that satisfies the specification.

Example

We illustrate the concepts in partial-program synthesis using the example in Figure 1. It is a pseudo code for a producer thread in a producer-consumer concurrent program. In this example, a thread must authenticate itself (using a password provided by the user) before starting to access shared data. After authentication a producer thread, in a loop, attempts to find an empty slot in a buffer and to store data in this slot. A consumer thread operates analogously.

First, consider the code concerned with accessing the shared data structure (lines 9 to 23), which in this case is a two-element buffer, modeled by variables `x` and `y`. Notice that no synchronization primitives are used. Thus, data races can occur. For example, two producer threads running concurrently might both determine that the variable `x` is empty, and might try to store values concurrently into `x`, leading to a loss of data. This is prevented by adding a declarative specification, $spec_1$, that requires the implementation to have no data races (the specification could also require that there should be no deadlocks, and that every acquired lock must be eventually released). This is sufficient to specify the task for the synthesizer — the synthesizer would need to place synchronization primitives in the code in order to make sure that the specification holds. However, the programmer might want to guide the synthesizer further, by explicitly stating which synchronization primitives should be used, and where in the code should they be placed. The synchronization primitives available to the synthesizer are specified using the `choicedef` command. The programmer also specifies where in the code synchronization operations are performed: this is done by `choice` statements in lines 11, 14, 16, 17, 20, and 22. Here, the choice statement can be replaced by any of the commands listed in the definition of C1. The programmer thus specifies that the synchronization will be performed using a global lock `gl` and variable-local locks `xl` and `yl`.

Second, consider the code concerned with the authentication (lines 1 to 8). It gives the user n possibilities to enter/guess the password. The programmer left the choice of n partially unspecified — this is the purpose of the `choice` statement on line 1. However, the programmer wants to make sure that the secrecy of the password is sufficiently protected, and in particular, that the password is not leaked by allowing too many login attempts. Let us assume that the programmer adds (naively) a specification $spec_2$ that requires that there is *no* information flow from the variable `passw` to the other program variables.

The task of the synthesizer is to construct a program, in such a way that the declarative specification, i.e., the conjunction of $spec_1$ and $spec_2$, is satisfied.

Consider the specification $spec_1$. Analyzing the partial program in Figure 1, we see that it allows three types of implementations. We describe a representative of each type.

```
choicedef C1 : {gl.lock(); xl.lock(); yl.lock();
             gl.unlock(); xl.unlock(); yl.unlock();skip;}

  public boolean produce() {
1:  n = choice(0..10);
2:  logged_in = false;
3:  for i:= 1 to n do {
4:    guess=read();
5:    if passw == inp then {
6:     logged_in = true;
7:     break }
8:  }
9:  if (logged_in) {
10:   while (true) {
11:     choice C1; //should be gl.lock() or xl.lock()
12      if is_empty(x) {
13:       x = compute_new_data();
14:       choice C1; //should be gl.unlock() or xl.unlock()
15:       return true }
16:     choice C1; //should be skip or xl.unlock()
17:     choice C1; //should be skip or yl.lock()
18:     if is_empty(y) {
19:       y = compute_new_data();
20:       choice C1; //should be gl.unlock() or yl.unlock()
21:       return true }
22:     choice C1; //should be gl.unlock() or  yl.unlock()
23:     return false } }
   }
```

Figure 1: Authenticated Producer-Consumer

- Incorrect implementations: implementation S_1 that does not use any locks. This leads to data races on both `x` and `y`.
- Global locking: implementation S_2 obtained by choosing `gl.lock()` at line 11, `gl.unlock()` at line 14, 20, and 22, and `skip` at the other `choice` locations.
- Variable-local locking: implementation S_3 obtained by choosing `xl.lock()` at line 11, choosing `xl.unlock()` at lines 14 and 16, choosing `yl.lock()` at line 17, and choosing `yl.unlock()` at lines 20 and 22.

The synthesizer could therefore choose freely from correct solutions such as S_1 and S_2.

Let us consider specification $spec_2$, which requires no information flow from the secure variable `passw` to the other variables. As it can be seen that allowing even one login attempt leaks at least one bit of information, the only possibility for the synthesizer is to allow no login attempts at all (i.e., the synthesizer chooses 0 at line 1).

Partial programs

A *partial program* is a nondeterministic program. *Synthesis locations* in a partial program are program locations which allow nondeterminism (i.e., locations from which the execution can continue in more than one way). For example, line number 1 in Figure 1 corresponds to a synthesis location. The set of synthesis locations can be defined implicitly or explicitly. The programmer can state that the synthesizer can add code at any program location (i.e., implicitly all locations are synthesis locations). Alternatively, the programmer can explicitly define which locations are synthesis locations, and leave only a small number of nondeterministic choices for the synthesizer. The second option allows the programmer to have better control over the resulting program, and allows for more efficient synthesis. It can be used for instance when the programmer has a good idea of the overall structure of the program, but is unsure about some complex details. For example, in the program in Figure 1, the programmer coded imperatively the functionality of accessing a data structure, and knew that synchronization will be achieved using locks, but left the details of lock placement to the synthesizer.

A program P is *allowed* by a partial program R, if P resolves the nondeterministic choices at all synthesis locations of R.

Boolean partial-program synthesis problem statement. Given a partial program R and a specification φ, either generate a program P such that P is allowed by R and P satisfies the specification φ, or report that such a program does not exist.

Algorithms for boolean partial-program synthesis

The boolean partial-program synthesis problem can be seen equivalently as a two-player graph game, where Player 1 resolves nondeterministic choices in synthesis locations, and Player 2 can choose inputs (and the schedule in the case of concurrent systems). Objectives of Players 1 and 2 are derived from the specification φ of the given boolean partial-program synthesis problem. The ω-regular objectives are a broad and widely studied class of objectives that capture many specifications occurring in practice. For finite-state systems and ω-regular objectives, existence of strategies for both players is decidable (see e.g. [24] for a survey of existing algorithms). Graph games have been used for synthesis in for example [27, 26, 23].

There is a number of other techniques (i.e., algorithmic techniques not based on graph games) for partial-program synthesis. These include counter-example guided inductive synthesis [37], synthesis from examples [25], or using the model generation capability of decision procedures [28].

3. QUANTITATIVE PARTIAL-PROGRAM SYNTHESIS

In this subsection, we show how the synthesis results can be improved by using quantitative measures and quantitative specifications.

First, we need to define a notion of a program quantity. A *program quantity* Q is a function that given a program returns a value D, where D is a total order. We consider program quantities that can arise in two different ways:

1. The quantity is inherently present in the semantics of the program. An example of a program quantity of this type is, in the context of a request-grant system, the proportion of requests that are dropped (i.e. not granted) by the system.

2. The quantity depends not only on the program semantics, but also on a resource dependent on the execution platform. Typical resources that are considered are memory, execution time, and power. An execution platform is modeled by a *resource model* that assigns costs to program operations. The cost of an operation need not be static. A stateful resource model can model history-dependent costs. If T is a resource model, we use the notation Q_T for the program quantity given by T.

Once we defined program quantities, we can generalize boolean partial-program synthesis in two different ways.

First, synthesis can use a *quantitative specification*, i.e., a specification that refers to a program quantity Q. This can often be seen as a relaxation of a correctness requirement. For example, in the context of a request-grant system, a boolean specification might require that all requests are granted. A quantitative specification might accept a system that drops fewer than 5% of requests. The articles [19] and [4] contain further examples and discussion of quantitative specifications.

Second, a program quantity Q can be used as a *quantitative objective*, that is a quantity which should be optimized (e.g. minimized or maximized) by the synthesis algorithm. For example, we might want a correct program with the smallest worst-case execution time, or the smallest memory consumption.

Quantitative partial-program synthesis problem statement. Given a partial program R, a (quantitative) specification φ, and a quantitative objective Q, either generate a program P such that $Q(P)$ is optimal over all programs that are allowed by R and satisfy φ, or report that such a program does not exist.

Example

Let us revisit the example in Figure 1.

First, consider the specification $spec_1$. We have remarked that without a quantitative requirement, the synthesizer could choose freely whether to return the implementation S_2 or S_3. However, these two implementations are not equivalent with respect to performance. The synthesizer can distinguish and choose between them using a resource model. For this example, let us consider history-free resource models that assign a cost l to locking, a cost c to operations that access the shared memory, and a cost b to thread-local operations. Consider two such resource models: model T that has the cost l far greater than the costs c and b, and model T' that has the cost c far greater than the other two costs. Under resource model T, the potential gain in performance due to more concurrency allowed by find-grained locking in implementation S_3 is outweighed by the cost of having to lock twice as often. This means that implementation S_2 will perform better in this case. On the other hand, under resource model T', the implementation S_3 will perform better (as allowing more concurrency yields greater benefits in this case) and should be the result of the synthesis. For quantitative synthesis, we thus use a quantitative objective Q_T (or $Q_{T'}$), in addition to the specification $spec_1$.

Second, consider specification $spec_2$. We have remarked that under this specification, the only solution to the synthesis problem is to allow no login attempts at all. We therefore relax this requirement to a quantitative specification $spec_2'$, that requires the information flow from the variable `passw` to the other program variables to be less than a constant d. (The details of the definition of quantitative information flow are not essential here — we refer the reader to [5] for more information.) Furthermore, we add a quantitative objective Q_n, which specifies that we prefer to allow as many login attempts as possible. Q_n is defined by the variable `n` of the example in Figure 1. The synthesizer should thus return the program with `n` maximized, while still keeping the required bound on information flow.

The specification for the whole program is therefore the

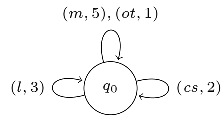

Figure 2: Resource model (l - locking cost, m - shared memory access cost, ot - cost of other operations).

conjunction of $spec_1$ and $spec_2$, while the quantitative objective is the pair of (Q_T, Q_n), with lexicographic ordering. (Note that Q_T and Q_n can be optimized independently.)

Program quantities

Many quantitative measures of programs can be useful in synthesis, and a number of them has already been used in the literature. We first list sample program quantities that depend only on the semantics of the program.

- Information flow from high security variables to the other program variables. See [17, 41, 5] for approaches to verification and synthesis of quantitative information flow properties.
- Relaxed correctness requirements (see e.g. [7]). We described an example of a request grant system. Instead of checking whether every request is granted, we might be interested in measuring how many requests are dropped. Similarly, linearizability, a standard correctness condition for concurrent programs can be relaxed to a quantitative notion of k-linearizability [32].
- Robustness to violations of input assumptions [3, 2]. For many systems, the execution should satisfy the specification or be "close" to satisfying the specification, even if the input assumptions are violated. This property is often referred to as graceful degradation.

The following program quantities do not depend only on the program semantics, but also on the execution platform. It is for modeling the execution platform that the resource model is used.

- Resource consumption (execution time, memory, power). We have mentioned resource models for measuring execution time. Similarly, resource models can be used to measure for example the maximum amount of memory used over an execution of the program, or the total energy used during an execution of the program. Models of a similar nature have been used in literature for analyzing worst-case execution time for programs [40] or for synthesizing power optimal schedulers [21]. The article [8] shows how resource consumption can be computed for programs composed from multiple components.
- Fault tolerance. In the context of a fault-tolerant distributed algorithm (see [29]), we might be interested in the number of nodes which can fail without compromising the correctness of the overall result.
- Concurrency. In the context of concurrent programs, program quantities that measure how many different interleavings are allowed might be of interest. Such measures include the size of atomic sections (considered for example in [39]). These measures can be seen as using an implicit resource model that prescribes that allowing more interleavings is better.

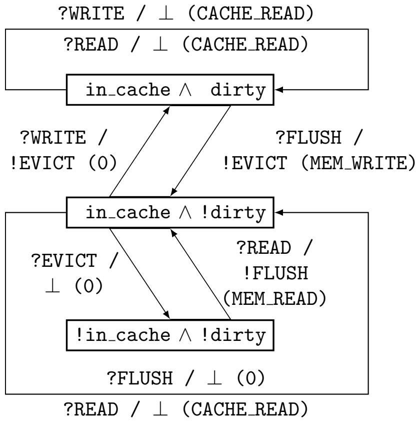

Figure 3: Resource model of a cache line

Given a resource model T and a program P, a program quantity $Q_T(P)$ is computed as follows. The resource model assigns a cost to every program operation. Quantity $Q_T(P)$ is then defined in two steps. First, the cost of a trace is defined. For programs with finite traces, one might obtain the value as a sum of the costs of the operations of the trace, if we are interested in e.g. the total amount of energy expended along a program execution. Other aggregating functions include maximum, average, etc. For programs with infinite traces, typically discounted sums or mean payoff functions are considered (see for example [10]). Second, given a cost of each trace, we can define $Q_T(P)$ as the worst-case (i.e., maximal or minimal value) over all the traces of the program, or the average-case (see for example [6]). A probability distribution over traces can be defined using a probability distribution over inputs (and over schedules, for a concurrent program).

Figures 2 and 3 contain two examples of resource models. These models were used in [6] for quantitative synthesis. Figure 2 has a one-state resource model for execution time analysis, similar to models T and T' discussed above. It assigns costs to locking, shared memory access, and all other operations. The effect of processor caches on execution time can be modeled by a simple resource model for caches. A cache line is modeled as in Figure 3. It assigns different costs to read and write actions depending on whether the line is cached. The full resource model is the synchronous product of one such automaton per memory line. The only actions in the resource model after the synchronous product (caches synchronize on `evict` and `flush`) are `READ` and `WRITE` actions. These actions are matched with the transitions of the partial program.

In the context of specification, verification, and synthesis, the term "quantitative" sometimes refers probabilistic settings. While we do not rule out probabilistic aspects (see for example [11]), we use the term quantitative much more broadly to refer to different types of program quantities. We also note that an important special case that can be viewed as a program quantity is real time. Real-time systems have been extensively studied. Synthesis [30] and games [20, 14] have been considered in this context. However, we want to emphasize that quantitative notions are useful also in the case of purely finite-state systems and specifications. We might for example be interested in how far a system is from a specification. Here the distance between a system and a specification intuitively measures how much we have to relax the specification so that it holds for the system [7].

Quantitative synthesis algorithms

We have mentioned that graph games can be used as a foundation for reactive synthesis. This foundation can be naturally extended to quantitative synthesis. The graph game will again be a two-player game where strategies of Player 1 correspond to programs. The optimal strategy then gives rise to the program that is the result of synthesis. Recall that the input to the quantitative partial program synthesis problem consists of a partial program R, a specification φ, and a quantitative objective Q. After the reduction to a graph game, φ will result in a qualitative objective, while Q will be translated into a quantitative objective of the game. There exist algorithms for solving finite-state games with quantitative objectives (such as mean payoff or discounted sum objectives) [42, 34], as well as algorithms for games with both qualitative (ω-regular) and quantitative objectives [13, 9]. Synthesis based on algorithms for graph games with quantitative objectives for finite state system has been explored recently [3, 12, 6] .

There exist several other techniques (i.e., algorithmic techniques not based on graph games) for synthesis with quantitative objectives. Emmi et al. reduce the optimal lock allocation problem to a 0-1 ILP which minimizes the conflict cost between atomic sections while simultaneously minimizing the number of locks [22]. Cherem et al. transform programs with pessimistic atomic sections to programs with locks. The transformation chooses from several granularities, using fine-grained locks if possible [15]. The approach by Vechev, Yahav, and Yorsh [39] tries to minimize the size of atomic section in concurrent programs. It is based on detecting invalid interleavings in the abstracted program. Whenever an abstract invalid interleaving is detected, either the abstraction is refined, or the program is modified by extending the atomic sections (and thereby eliminating the invalid interleaving).

Future directions

We argued that quantitative specifications and objectives are needed in partial-program synthesis in order to produce optimal programs, while enabling simpler specifications. We therefore believe that quantitative specifications and objectives are needed if the synthesis approach is to succeed in practice. However, there are several challenges before quantitative synthesis tools can become more widely applicable.

First, there is a programming languages challenge to find languages that can capture various specification artifacts. The programmer needs to be able to define a partial program with constraints on its nondeterminism, a (quantitative) specification, and a quantitative objective. In order for programmers to benefit from synthesis techniques, it is necessary to find a way for integrated specification of these synthesis inputs.

Second, consider resource-aware synthesis, that is, synthesis that returns programs that are optimal with respect to the consumption of a particular resource. In this case, it is necessary to have resource models that capture target architectures on which the program is to be used. The models do not need to be designed for each synthesis problem, but instead only once for each system architecture. However, the problem of constructing resource models and finding their parameters remains a research challenge.

Third, there is an important issue of scalability. Quantitative synthesis is useful for well-defined classes of small programs (e.g. the class of concurrent data structures), where current techniques and algorithms might be sufficient. However, new approaches are needed if quantitative synthesis is to scale. The techniques could include quantitative abstractions of games, i.e., abstractions that preserve values of strategies with respect to quantitative objectives (or at least the relative ordering of strategies with respect to quantitative objectives). Further improvements or development of new (symbolic) quantitative game solvers would also improve the scalability of synthesis.

4. REFERENCES

[1] A. Biermann. Automatic programming: A tutorial on formal methodologies. *J. Symb. Comput.*, 1(2):119–142, 1985.

[2] R. Bloem, K. Chatterjee, K. Greimel, T. Henzinger, B. Jobstmann, and R. Singh. Specification-centered robustness. In *SIES*, pages 176–185, 2011.

[3] R. Bloem, K. Chatterjee, T. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, 2009.

[4] U. Boker, K. Chatterjee, T. Henzinger, and O. Kupferman. Temporal specifications with accumulative values. In *LICS*, 2011.

[5] P. Černý, K. Chatterjee, and T. Henzinger. The complexity of quantitative information flow problems. In *CSF*, pages 205–218, 2011.

[6] P. Černý, K. Chatterjee, T. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In *CAV*, pages 243–259, 2011.

[7] P. Černý, T. Henzinger, and A. Radhakrishna. Simulation distances. In *CONCUR*, pages 253–268, 2010.

[8] A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource interfaces. In *EMSOFT*, pages 117–133, 2003.

[9] K. Chatterjee and L. Doyen. Energy parity games. In *ICALP (2)*, pages 599–610, 2010.

[10] K. Chatterjee, L. Doyen, and T. Henzinger. Quantitative languages. *ACM Trans. Comput. Log.*, 11(4), 2010.

[11] K. Chatterjee, T. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV*, pages 380–395, 2010.

[12] K. Chatterjee, T. Henzinger, B. Jobstmann, and R. Singh. Quasy: Quantitative synthesis tool. In *TACAS*, pages 267–271, 2011.

[13] K. Chatterjee, T. Henzinger, and M. Jurdzinski. Mean-payoff parity games. In *LICS*, pages 178–187, 2005.

[14] K. Chatterjee, T. Henzinger, and V. Prabhu. Timed parity games: Complexity and robustness. In *FORMATS*, pages 124–140, 2008.

[15] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*, pages 304–315, 2008.

[16] A. Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*, 1962.

[17] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[18] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, 1981.

[19] L. de Alfaro, M. Faella, T. Henzinger, R. Majumdar, and M. Stoelinga. Model checking discounted temporal properties. *Theor. Comput. Sci.*, 345(1):139–170, 2005.

[20] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *CONCUR*, pages 142–156, 2003.

[21] L. de Alfaro, V. Raman, M. Faella, and R. Majumdar. Code aware resource management. In *EMSOFT*, pages 191–202, 2005.

[22] M. Emmi, J. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL*, pages 291–296, 2007.

[23] E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *CAV*, pages 263–277, 2009.

[24] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[25] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[26] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *CAV*, pages 258–262, 2007.

[27] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[28] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[29] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1997.

[30] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.

[31] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[32] H. Payer, H. Röck, C. Kirsch, and A. Sokolova. Scalability versus semantics of concurrent fifo queues. In *PODC*, pages 331–332, 2011.

[33] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.

[34] A. Puri. *Theory of Hybrid Systems and Discrete Event Systems.* PhD thesis, EECS Department, University of California, Berkeley, 1995.

[35] D. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Software Eng.*, 16(9):1024–1043, 1990.

[36] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[37] A. Solar-Lezama, C. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, 2008.

[38] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[39] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[40] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm. Static timing analysis for hard real-time systems. In *VMCAI*, 2010.

[41] H. Yasuoka and T. Terauchi. Quantitative information flow - verification hardness and possibilities. In *CSF*, pages 15–27, 2010.

[42] U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theor. Comput. Sci.*, 158(1&2):343–359, 1996.