Assume-Guarantee Synthesis for Concurrent Reactive Programs with Partial Information*

Roderick Bloem[1], Krishnendu Chatterjee[2], Swen Jacobs[1,3], and Robert Könighofer[1]

[1] IAIK, Graz University of Technology, Austria
[2] IST Austria (Institute of Science and Technology Austria)
[3] Reactive Systems Group, Saarland University, Germany

Abstract. Synthesis of program parts is particularly useful for concurrent systems. However, most approaches do not support common design tasks, like modifying a single process without having to re-synthesize or verify the whole system. Assume-guarantee synthesis (AGS) provides robustness against modifications of system parts, but thus far has been limited to the perfect information setting. This means that local variables cannot be hidden from other processes, which renders synthesis results cumbersome or even impossible to realize. We resolve this shortcoming by defining AGS under partial information. We analyze the complexity and decidability in different settings, showing that the problem has a high worst-case complexity and is undecidable in many interesting cases. Based on these observations, we present a pragmatic algorithm based on bounded synthesis, and demonstrate its practical applicability on several examples.

1 Introduction

Concurrent programs are notoriously hard to get right, due to unexpected behavior emerging from the interaction of different processes. At the same time, concurrency aspects such as mutual exclusion or deadlock freedom are easy to express declaratively. This makes concurrent programs an ideal subject for automatic synthesis. Due to the prohibitive complexity of synthesis tasks [33,34,17], the automated construction of *entire* programs from high-level specifications such as LTL is often unrealistic. More practical approaches are based on partially implemented programs that should be completed or refined automatically [17,16,39], or program repair, where suitable replacements need to be synthesized for faulty program parts [25]. This paper focuses on such applications, where parts of the system are already given.

When several processes need to be synthesized or refined simultaneously, a fundamental question arises: What are the assumptions about the behavior of other processes on which a particular process should rely? The classical synthesis approaches assume either completely adversarial or cooperative behavior, which leads to problems in both

* This work was supported by the Austrian Science Fund (FWF) through the research network RiSE (S11406-N23, S11407-N23) and grant nr. P23499-N23, by the European Commission through an ERC Start grant (279307: Graph Games) and project STANCE (317753), as well as by the German Research Foundation (DFG) through SFB/TR 14 AVACS and project ASDPS (JA 2357/2-1).

cases: adversarial components may result in unrealizability of the system, while cooperative components may rely on a specific form of cooperation, and therefore are not robust against even small changes in a single process. Assume-Guarantee Synthesis (AGS) [9] uses a more reasonable assumption: processes are adversarial, but will not violate their own specification to obstruct others. Therefore, a system constructed by AGS will still satisfy its overall specification if we replace or refine one of the processes, as long as the new process satisfies its local specification. Furthermore, AGS leads to the desired solutions in cases where the classical notions (of cooperative or completely adversarial processes) do not, for example in the synthesis of mutual exclusion protocols [9] or fair-exchange protocols for digital contract signing [13].

A drawback of existing algorithms for AGS [9,13] is that they only work in a perfect information setting. This means that each component can access and use the values of all variables of the other processes. This is a major restriction, as most concurrent implementations rely on variables that are *local* to one process, and should not be changed or observed by the other process. While classical notions of synthesis have been considered in such partial information settings before [28,17], we provide the first solution for AGS with partial information.

Contributions. In this work, we extend assume-guarantee synthesis to the synthesis of processes with partial information. In particular:

i) We analyze the complexity and decidability of AGS by reductions to games with three players. We distinguish synthesis problems based on informedness (perfect or partial) and resources (bounded or unbounded memory) of processes, and on specifications from different fragments of linear-time temporal logic (LTL).
ii) In light of the high complexity of many AGS problems, we propose a pragmatic approach, based on partially implemented programs and synthesis with bounded resources. We extend the bounded synthesis approach [18] to enable synthesis from partially defined, non-deterministic programs, and to the AGS setting.
iii) We provide the first implementation of AGS, integrated into a programming model that allows for a combined imperative-declarative programming style with fine-grained, user-provided restrictions on the exchange of information between processes. To obtain efficient and simple code, our prototype also supports optimization of the synthesized program with respect to some basic user-defined metrics.
iv) We demonstrate the value of our approach on a number of small programs and protocols, including Peterson's mutual exclusion protocol, a double buffering protocol, and synthesis of atomic sections in a concurrent device driver. We also demonstrate how the robustness of AGS solutions allows us to refine parts of the synthesized program without starting synthesis from scratch.

2 Motivating Example

We illustrate our approach using the running example of [9], a version of Peterson's mutual exclusion protocol.

Sketch. We use the term *sketch* for concurrent reactive programs with non-deterministic choices. Listing 1 shows a sketch for Peterson's protocol with processes P_1 and P_2.

Listing 1: Sketch of Peterson's mutual exclusion protocol. F=false, T=true.

```
0                                          turn:=F; flag1:=F; flag2:=F;
1  cr1:=F; wait1:=F;                    21 cr2:=F; wait2:=F;
2  do { // Process P1:                  22 do { // Process P2:
3     flag1:=T;                         23    flag2:=T;
4     turn:=T;                          24    turn:=F;
5     while(?1,1) {} //wait             25    while(?2,1) {} //wait
6     cr1:=T;                           26    cr2:=T; //read:=?2,3
7     cr1:=F; flag1:=F; wait1:=T;       27    cr2:=F; flag2:=F; wait2:=T;
8     while(?1,2) {} //local work       28    while(?2,2) {} //local work
9     wait1:=F;                         29    wait2:=F;
10 } while(T)                           30 } while(T)
```

Variable `flag`i indicates that P_i wants to enter the critical section, and `cr`i that P_i is in the critical section. The first `while`-loop waits for permission to enter the critical section, the second loop models some local computation. Question marks denote non-deterministic choices, and we want to synthesize expressions that replace question marks such that P_1 and P_2 never visit the critical section simultaneously.

Specification. The desired properties of both processes are that (1) whenever a process wants to enter the critical section, it will eventually enter it (starvation freedom), and (2) the two processes are never in the critical section simultaneously (mutual exclusion). In LTL[1], the specification is $\Phi_i = \mathsf{G}(\neg\texttt{cr1} \vee \neg\texttt{cr2}) \wedge \mathsf{G}(\texttt{flag}i \rightarrow \mathsf{F}\,\texttt{cr}i)$, for $i \in \{1, 2\}$.

Failure of classical approaches. There are essentially two options for applying standard synthesis techniques. First, we may assume that both processes are cooperative, and synthesize all $?_{i,j}$ simultaneously. However, the resulting implementation of P_2 may only work for the computed implementation of P_1, i.e., changing P_1 may break P_2. For instance, the solution $?_{1,1}$ = `turn & flag2`, $?_{2,1}$ = `!turn` and $?_{i,2}$ = `F` satisfies the specification, but changing $?_{1,2}$ in P_1 to `T` will make P_2 starve. Note that this is not just a hypothetical case; we got exactly this solution in our experiments. As a second option, we may assume that the processes are adversarial, i.e., P_2 must work for *any* P_1 and vice versa. However, under this assumption, the problem is unrealizable [9].

Success of Assume-Guarantee Synthesis (AGS) [9]. AGS fixes this dilemma by requiring that P_2 must work for *any* realization of P_1 that satisfies its local specification (and vice versa). An AGS solution for Listing 1 is $?_{1,1}$ = `turn & flag2`, $?_{2,1}$ = `!turn & flag2` and $?_{i,2}$ = `F` for $i \in \{1, 2\}$.

Added advantage of AGS. If one process in an AGS solution is changed or extended, but still satisfies its original specification, then the other, unchanged process is guaranteed to remain correct as well. We illustrate this feature by extending P_2 with a new variable named `read`. It is updated in a yet unknown way (expressed by $?_{2,3}$) whenever P_2 enters the critical section in line 26 of Listing 1. Assume that we want to implement $?_{2,3}$ such that `read` is true and false infinitely often. We take the solution from the previous paragraph and synthesize $?_{2,3}$ such that P_2 satisfies $\Phi_2 \wedge (\mathsf{G}\,\mathsf{F}\,\neg\texttt{read}) \wedge (\mathsf{G}\,\mathsf{F}\,\texttt{read})$, where Φ_2 is the original specification of P_2. The fact that the modified process still satisfies Φ_2 implies that P_1 will still satisfy its original specification. We also notice that

[1] In case the reader is not familiar with LTL: G is a temporal operator meaning "in all time steps"; likewise F means "at some point in the future".

Listing 2: Result for Listing 1: `turn` is replaced by memory **m** in a clever way.

```
0                                  flag1:=F; flag2:=F; m:=F;
1  cr1:=F; wait1:=F;                                  21 cr2:=F; wait2:=F;
2  do { // Process P1:                                22 do { // Process P2:
3     flag1:=T;                                       23    flag2:=T;
4     while(!m) {} //wait                             24    while(m) {} //wait
5     cr1:=T;                                         25    cr2:=T;
6     cr1:=F; flag1:=F; wait1:=T;                     26    cr2:=F; flag2:=F; wait2:=T;
7     while(input1()) //work                          27    while(input2()) //work
8        m:=F;                                        28       m:=T;
9     wait1:=F; m:=F;                                 29    wait2:=F; m:=T;
10 } while(T)                                         30 } while(T)
```

modular refinement saves overall synthesis time: our tool takes $19 + 55 = 74$ seconds to first synthesize the basic AGS solution for both processes and then refine P_2 in a second step to get the expected solution with $?_{\mathbf{2,3}} = \neg\texttt{read}$, while direct synthesis of the refined specification for both processes requires 263 seconds.

Drawbacks of the existing AGS framework [9]. While AGS provides important improvements over classical approaches, it may still produce solutions like $?_{\mathbf{1,1}} = \texttt{turn} \wedge \neg\texttt{wait2}$ and $?_{\mathbf{2,1}} = \neg\texttt{turn} \wedge \neg\texttt{wait1}$. However, `wait2` is intended to be a *local* variable of P_2, and thus invisible for P_1. Solutions may also utilize modeling artifacts such as program counters, because AGS has no way to restrict the information visible to other processes. As a workaround, the existing approach [9] allows the user to define candidate implementations for each **?**, and let the synthesis algorithm select one of the candidates. However, when implemented this way, a significant part of the problem needs to be solved by the user.

AGS with partial information. Our approach resolves this shortcoming by allowing the declaration of local variables. The user can write $f_{1,1}(\texttt{turn}, \texttt{flag2})$ instead of $?_{1,1}$ to express that the solution may only depend on `turn` and `flag2`. Including more variables of P_1 does not make sense for this example, because their value is fixed at the call site. When setting $?_{\mathbf{2,1}} = f_{1,2}(\texttt{turn}, \texttt{flag1})$ (and $?_{\mathbf{i,2}} = f_{i,2}()$), we get the solution proposed by Peterson: $?_{\mathbf{1,1}} = \texttt{turn} \wedge \texttt{flag2}$ and $?_{\mathbf{2,1}} = \neg\texttt{turn} \wedge \texttt{flag1}$ (and $?_{\mathbf{i,2}} = \texttt{F}$). This is the only AGS solution with these dependency constraints.

AGS with additional memory and optimization. Our approach can also introduce additional memory in form of new variables. As with existing variables, the user can specify which question mark may depend on the memory variables, and also which variables may be used to update the memory. For our example, this feature can be used to synthesize the entire synchronization from scratch, without using `turn`, `flag1`, and `flag2`. Suppose we remove `turn`, allow some memory `m` instead, and impose the following restrictions: $?_{1,1} = f_{1,1}(\texttt{flag2}, \texttt{m})$, $?_{2,1} = f_{2,1}(\texttt{flag1}, \texttt{m})$, $?_{i,2}$ is an uncontrollable input (to avoid overly simplistic solutions), and `m` can only be updated depending on the program counter and the old memory content. Our approach also supports cost functions over the result, and optimizes solutions iteratively. For our example, the user can assign costs for each memory update in order to obtain a simple solution with few memory updates. In this setup, our approach produces the solution presented in Listing 2. It is surprisingly simple: It requires only one bit of memory `m`,

ignores both `flags` (although we did not force it to), and updates `m` only twice[2]. Our proof-of-concept implementation took only 74 seconds to find this solution.

3 Definitions

In this section we first define processes, refinement, schedulers, and specifications. Then we consider different versions of the co-synthesis problem, depending on *informedness* (partial or perfect), *cooperation* (cooperative, competitive, assume-guarantee), and *resources* (bounded or unbounded) of the players.

Variables, valuations, traces. Let X be a finite set of binary variables. A *valuation* on X is a function $v : X \to \mathbb{B}$ that assigns to each variable $x \in X$ a value $v(x) \in \mathbb{B}$. We write $\mathbb{B}^X$ for the set of valuations on X, and $u \circ v$ for the concatenation of valuations $u \in \mathbb{B}^X$ and $v \in \mathbb{B}^{X'}$ to a valuation in $\mathbb{B}^{X \cup X'}$. A *trace* on X is an infinite sequence $(v_0, v_1, \ldots)$ of valuations on X. Given a valuation $v \in \mathbb{B}^X$ and a subset $X' \subseteq X$ of the variables, define $v{\restriction_{X'}}$ as the *restriction* of v to X'. Similarly, for a trace $\pi = (v_0, v_1, \ldots)$ on X, write $\pi{\restriction_{X'}} = (v_0{\restriction_{X'}}, v_1{\restriction_{X'}}, \ldots)$ for the restriction of π to the variables X'. The restriction operator extends naturally to sets of valuations and traces.

Processes and refinement. We consider non-deterministic processes, where the non-determinism is modeled by variables that are not under the control of the process. We call these variables *input*, but they may also be internal variables with non-deterministic updates. For $i \in \{1, 2\}$, a *process* $P_i = (X_i, O_i, Y_i, \tau_i)$ consists of finite sets

- X_i of modifiable state variables,
- $O_i \subseteq X_{3-i}$ of observable (but not modifiable) state variables,
- Y_i of input variables,

and a *transition function* $\tau_i : \mathbb{B}^{X_i} \times \mathbb{B}^{O_i} \times \mathbb{B}^{Y_i} \to \mathbb{B}^{X_i}$. The transition function maps a current valuation of state and input variables to the next valuation for the state variables. We write $X = X_1 \cup X_2$ for the set of state variables of both processes, and similarly $Y = Y_1 \cup Y_2$ for the input variables. Note that some variables may be shared by both processes. Variables that are not shared between processes will be called *local* variables.

We obtain a refinement of a process by resolving some of the non-determinism introduced by input variables, and possibly extending the sets of local state variables. Formally, let $C_i \subseteq Y_i$ be a set of *controllable variables*, let $Y_i' = Y_i \setminus C_i$, and let $X_i' \supseteq X_i$ be an extended (finite) set of state variables, with $X_1' \cap X_2' = X_1 \cap X_2$. Then a *refinement* of process $P_i = (X_i, O_i, Y_i, \tau_i)$ *with respect to* C_i is a process $P_i' = (X_i', O_i, Y_i', \tau_i')$ with a transition function $\tau_i' : \mathbb{B}^{X_i'} \times \mathbb{B}^{O_i} \times \mathbb{B}^{Y_i'} \to \mathbb{B}^{X_i'}$ such that for all $\overline{x} \in \mathbb{B}^{X_i'}, \overline{o} \in \mathbb{B}^{O_i}, \overline{y} \in \mathbb{B}^{Y_i'}$ there exists $\overline{c} \in \mathbb{B}^{C_i}$ with

$$\tau_i'(\overline{x}, \overline{o}, \overline{y}){\restriction_{X_i}} = \tau_i(\overline{x}{\restriction_{X_i}}, \overline{o}, \overline{y} \circ \overline{c}).$$

We write $P_i' \preceq P_i$ to denote that P_i' is a refinement of P_i.

[2] The memory `m` is updated whenever an input is read in line 7 or 27; we copied the update into both branches to increase readability.

Important modeling aspects. Local variables are used to model partial information: all decisions of a process need to be independent of the variables that are local to the other process. Furthermore, variables in $X_i' \setminus X_i$ are used to model additional memory that a process can use to store observed information. We say a refinement is *memoryless* if $X_i' = X_i$, and it is *b-bounded* if $|\, X_i' \setminus X_i \,| \leq b$.

Schedulers, executions. A *scheduler* for processes P_1 and P_2 chooses at each computation step whether P_1 or P_2 can take a step to update its variables. Let $\mathcal{X}_1, \mathcal{X}_2$ be the sets of all variables (state, memory, input) of P_1 and P_2, respectively, and let $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$. Let furthermore $V = \mathbb{B}^{\mathcal{X}}$ be the set of global valuations. Then, the scheduler is a function $\mathsf{sched} : V^* \to \{1, 2\}$ that maps a finite sequence of global valuations to a process index $i \in \{1, 2\}$. Scheduler sched is *fair* if for all traces $(v_0, v_1, \ldots) \in V^\omega$ it assigns infinitely many turns to both P_1 and P_2, i.e., there are infinitely many $j \geq 0$ such that $\mathsf{sched}(v_0, \ldots, v_j) = 1$, and infinitely many $k \geq 0$ such that $\mathsf{sched}(v_0, \ldots, v_k) = 2$.

Given two processes P_1, P_2, a scheduler sched, and a start valuation v_0, the set of possible *executions* of the parallel composition $P_1 \parallel P_2 \parallel \mathsf{sched}$ is

$$[\![P_1 \parallel P_2 \parallel \mathsf{sched}, v_0]\!] = \left\{ (v_0, v_1, \ldots) \in V^\omega \;\middle|\; \begin{array}{l} \forall j \geq 0.\ \mathsf{sched}(v_0, v_1, \ldots, v_j) = i \\ \text{and } v_{j+1}{\restriction}_{(\mathcal{X} \setminus \mathcal{X}_i)} = v_j{\restriction}_{(\mathcal{X} \setminus \mathcal{X}_i)} \\ \text{and } v_{j+1}{\restriction}_{\mathcal{X}_i \setminus Y_i} \in \tau_i(v_j{\restriction}_{\mathcal{X}_i}) \end{array} \right\}.$$

That is, at every turn the scheduler decides which of the processes makes a transition, and the state and memory variables are updated according to the transition function of that process. Note that during turns of process P_i, the values of local variables of the other process (in $\mathcal{X} \setminus \mathcal{X}_i$) remain unchanged.

Safety, GR(1), LTL. A specification Φ is a set of traces on $X \cup Y$. We consider ω-regular specifications, in particular the following fragments of LTL:[3]

- *safety properties* are of the form $\mathsf{G}\, B$, where B is a Boolean formula over variables in $X \cup Y$, defining a subset of valuations that are safe.
- *GR(1) properties* are of the form $\left(\bigwedge_i \mathsf{G}\,\mathsf{F}\, L_e^i\right) \rightarrow \left(\bigwedge_j \mathsf{G}\,\mathsf{F}\, L_s^j\right)$, where the L_e^i and L_s^j are Boolean formulas over $X \cup Y$.
- *LTL properties* are given as arbitrary LTL formulas over $X \cup Y$. They are a subset of the ω-regular properties.

Co-Synthesis. In all co-synthesis problems, the input to the problem is given as: two processes P_1, P_2 with $P_i = (X_i, O_i, Y_i, \tau_i)$, two sets C_1, C_2 of controllable variables with $C_i \subseteq Y_i$, two specifications Φ_1, Φ_2, and a start valuation $v_0 \in \mathbb{B}^{X \cup Y}$, where $Y = Y_1 \cup Y_2$.

Cooperative co-synthesis. The *cooperative co-synthesis problem* is to find out whether there exist two processes $P_1' \preceq P_1$ and $P_2' \preceq P_2$, and a valuation v_0' with $v_0'{\restriction}_{X \cup Y} = v_0$, such that for all fair schedulers sched we have

$$[\![P_1' \parallel P_2' \parallel \mathsf{sched}, v_0']\!]{\restriction}_{X \cup Y} \subseteq \Phi_1 \wedge \Phi_2.$$

[3] For a definition of syntax and semantics of LTL, see e.g. [15].

Competitive co-synthesis. The *competitive co-synthesis problem* is to determine whether there exist two processes $P_1' \preceq P_1$ and $P_2' \preceq P_2$, and a valuation v_0' with $v_0' \restriction_{X \cup Y} = v_0$, such that for all fair schedulers sched we have

(i) $[\![P_1' \parallel P_2 \parallel \mathsf{sched}, v_0']\!] \restriction_{X \cup Y} \subseteq \Phi_1$, and
(ii) $[\![P_1 \parallel P_2' \parallel \mathsf{sched}, v_0']\!] \restriction_{X \cup Y} \subseteq \Phi_2$.

Assume-guarantee synthesis. The *assume-guarantee synthesis (AGS) problem* is to determine whether there exist two processes $P_1' \preceq P_1$ and $P_2' \preceq P_2$, and a valuation v_0' with $v_0' \restriction_{X \cup Y} = v_0$, such that for all fair schedulers sched we have

(i) $[\![P_1' \parallel P_2 \parallel \mathsf{sched}, v_0']\!] \restriction_{X \cup Y} \subseteq \Phi_2 \rightarrow \Phi_1$,
(ii) $[\![P_1 \parallel P_2' \parallel \mathsf{sched}, v_0']\!] \restriction_{X \cup Y} \subseteq \Phi_1 \rightarrow \Phi_2$, and
(iii) $[\![P_1' \parallel P_2' \parallel \mathsf{sched}, v_0']\!] \restriction_{X \cup Y} \subseteq \Phi_1 \wedge \Phi_2$.

We refer the reader to [9] for more intuition and a detailed discussion of AGS.

Informedness and boundedness. A synthesis problem is under *perfect information* if $X_i \cup O_i = X$ for $i \in \{1, 2\}$, and $Y_1 = Y_2$. That is, both processes have knowledge about all variables in the system. Otherwise, it is under *partial information*. A synthesis problem is *memoryless* (or b-bounded) if we additionally require that P_1', P_2' are memoryless (or b-bounded) refinements of P_1, P_2.

Optimization criteria. Let $\mathcal{P}$ be the set of all processes. A cost function is a function $\mathsf{cost} : \mathcal{P} \times \mathcal{P} \rightarrow \mathbb{N}$ that assigns a cost to a tuple of processes. By requiring that the cost of solutions is minimal or below a certain threshold, we will use cost functions to optimize synthesis results.

Note on robustness against modifications. Suppose P_1', P_2' are the result of AGS on a given input, including specifications Φ_1, Φ_2. By the properties of AGS, this solution is robust against replacing one of the processes, say P_2, with a different solution: if a replacement P_2'' of P_2' satisfies Φ_2, then the overall system will still be correct. If we furthermore ensure that conditions (ii) and (iii) of AGS are satisfied by P_1' and P_2'', then this pair is again an AGS solution, i.e., we can go on and refine another process.

Co-synthesis of more than 2 *processes.* The definitions above naturally extend to programs with more than 2 concurrent processes, cp. [13] for AGS with 3 processes.

4 Complexity and Decidability of AGS

We give an an overview of the complexity of AGS. The complexity results are with respect to the size of the input, where the input consists of the given non-deterministic state transition system and the specification formula (i.e., the size of the input is the size of the explicit state transition system and the length of the formula).

Theorem 1. *The complexity of AGS is given in the following table:*

	Bounded Memory		*Unbounded Memory*	
	Perfect Inf.	*Partial Inf.*	*Perfect Inf.*	*Partial Inf.*
Safety	*P*	*NP-C*	*P*	*Undec*
GR(1)	*NP-C*	*NP-C*	*P*	*Undec*
LTL	*PSPACE-C*	*PSPACE-C*	*2EXP-C*	*Undec*

Note that the complexity classes for memoryless AGS are the same as for AGS with bounded memory — the case of bounded memory reduces to the memoryless case, by considering a game that is larger by a constant factor: the given bound.

Also note that if we consider the results in the order given by the columns of the table, they form a non-monotonic pattern: (1) For safety objectives the complexity increases and then decreases (from PTIME to NP-complete to PTIME again); (2) for GR(1) objectives it remains NP-complete and finally decreases to PTIME; and (3) for LTL it remains PSPACE-complete and then increases to 2 EXPTIME-complete.

In the following, we give proof ideas for these complexity results. For formal definitions of three-player games, we refer the reader to [9].

Proof ideas.

First Column: Bounded Memory, Perfect Information.

Safety: It was shown in [9] that AGS solutions can be obtained from the solutions of games with secure equilibria. It follows from the results of [10] that for games with safety objectives, the solution for secure equilibria reduces to solving games with safety and reachability objectives for which memoryless strategies suffice (i.e., memoryless strategies are as powerful as arbitrary strategies for safety objectives). It also follows from [10] that for safety objectives, games with secure equilibria can be solved in polynomial time.

GR(1): It follows from the results of [20] that even in a graph (not a game) the question whether there exists a memoryless strategy to visit two distinct states infinitely often is NP-hard (a reduction from directed subgraph homeomorphism). Since visiting two distinct states infinitely often is a conjunction of two Büchi objectives, which is a special case of GR(1) objectives, the lower bound follows. For the NP upper bound, the witness memoryless strategy can be guessed, and once a memoryless strategy is fixed, we have a graph, and the polynomial-time verification procedure is the polynomial-time algorithm for model checking graphs with GR(1) objectives [32].

LTL: In the special case of a game graph where every player-1 state has exactly one outgoing edge, the memoryless AGS problem is an LTL model checking problem, and thus the lower bound of LTL model checking [15] implies PSPACE-hardness. For the upper bound, we guess a memoryless strategy (as for GR(1)), and the verification problem is an LTL model checking question. Since LTL model checking is in PSPACE [15] and NPSPACE=PSPACE (by Savitch's theorem) [37,30], we obtain the desired result.

Second Column: Bounded Memory, Partial Information.

Safety: The lower bound result was established in [12]. For the upper bound, again the witness is a memoryless strategy. Given the fixed strategy, we have a graph problem with safety and reachability objectives that can be solved in polynomial time (for the polynomial-time verification).

GR(1): The lower bound follows from the perfect-information case; for the upper bound, we can again guess and check a memoryless strategy.

LTL: Similar to the perfect information case, given above.

Third column: Unbounded Memory, Perfect Information.

Safety: As mentioned before, for AGS under perfect information and safety objectives, the memoryless and the general problem coincide, implying this result.

GR(1): It follows from results of [9,10] that solving AGS for perfect-information games requires solving games with implication conditions. Since games with implication of GR(1) objectives can be solved in polynomial time [21], the result follows.

LTL: The lower bound follows from standard LTL synthesis [33]. For the upper bound, AGS for perfect-information games requires solving implication games, and games with implication of LTL objectives can be solved in 2EXPTIME [33]. The desired result follows.

Fourth column: Unbounded Memory, Partial Information.

It was shown in [31] that three-player partial-observation games are undecidable, and it was also shown that the undecidability result holds for safety objectives too [11].

5 Algorithms for AGS

Given the undecidability of AGS in general, and its high complexity for most other cases, we propose a pragmatic approach that divides the general synthesis problem into a sequence of synthesis problems with a bounded amount of memory, and encodes the resulting problems into SMT formulas. Our encoding is inspired by the *Bounded Synthesis* approach [18], but supports synthesis from non-deterministic program sketches, as well as AGS problems. By iteratively deciding whether there exists an implementation for an increasing bound on the number of memory variables, we obtain a semi-decision procedure for AGS with partial information.

We first define the procedure for cooperative co-synthesis problems, and then show how to extend it to AGS problems.

5.1 SMT-based Co-Synthesis from Program Sketches

Consider a *cooperative* co-synthesis problem with inputs P_1 and P_2, defined as $P_i = (X_i, O_i, Y_i, \tau_i)$, two sets C_1, C_2 of controllable variables with $C_i \subseteq Y_i$, a specification $\Phi_1 \wedge \Phi_2$, and a start valuation $v_0 \in \mathbb{B}^{X \cup Y}$, where $Y = Y_1 \cup Y_2$.

In the following, we describe a set of SMT constraints such that a model represents refinements $P_1' \preceq P_1, P_2' \preceq P_2$ such that for all fair schedulers sched, we have $[\![P_1' \parallel P_2' \parallel \mathsf{sched}, v_0]\!] \subseteq \Phi_1 \wedge \Phi_2$. Assume we are given a bound $b \in \mathbb{N}$, and let Z_1, Z_2 be disjoint sets of additional memory variables with $|\, Z_i \,| = b$ for $i \in \{1, 2\}$.

Constraints on given transition functions. In the expected way, the transition functions τ_1 and τ_2 are declared as functions $\tau_i : \mathbb{B}^{X_i} \times \mathbb{B}^{O_i} \times \mathbb{B}^{Y_i} \to \mathbb{B}^{X_i}$, and directly encoded into SMT constraints by stating $\tau_i(\overline{x}, \overline{o}, \overline{y}) = \overline{x}'$ for every $\overline{x} \in \mathbb{B}^{X_i}, \overline{o} \in \mathbb{B}^{O_i}, \overline{y} \in \mathbb{B}^{Y_i}$, according to the given transition functions τ_1, τ_2.

Constraints for interleaving semantics, fair scheduling. To obtain an encoding for interleaving semantics, we add a scheduling variable s to both sets of inputs Y_1 and Y_2, and require that (i) $\tau_1(\overline{x}, \overline{o}, \overline{y}) = \overline{x}$ whenever $\overline{y}(s) = \mathsf{false}$, and (ii) $\tau_2(\overline{x}, \overline{o}, \overline{y}) = \overline{x}$ whenever $\overline{y}(s) = \mathsf{true}$. Fairness of the scheduler can then be encoded as the LTL formula $\mathsf{G}\,\mathsf{F}\, s \wedge \mathsf{G}\,\mathsf{F}\, \neg s$, abbreviated fair in the following.

Constraints on resulting strategy. Let $X_i' = X_i \cup Z_i$ be the extended state set, and $Y_i' = Y_i \setminus C_i$ the set of input variables of process P_i', reduced by its controllable

variables. Then the resulting strategy of P_i' is represented by functions $\mu_i : \mathbb{B}^{X_i'} \times \mathbb{B}^{O_i} \times \mathbb{B}^{Y_i'} \to \mathbb{B}^{Z_i}$ to update the memory variables, and $f_i : \mathbb{B}^{X_i'} \times \mathbb{B}^{O_i} \times \mathbb{B}^{Y_i'} \to \mathbb{B}^{C_i}$ to resolve the non-determinism for controllable variables. Functions f_i and μ_i for $i \in \{1, 2\}$ are constrained indirectly using constraints on an auxiliary annotation function that will ensure that the resulting strategy satisfies the specification $\Phi = (\mathsf{fair} \rightarrow \Phi_1 \wedge \Phi_2)$. To obtain these constraints, first transform Φ into a universal co-Büchi automaton $\mathcal{U}_\Phi = (Q, q_0, \Delta, F)$, where

- Q is a set of states and $q_0 \in Q$ is the initial state,
- $\Delta \subseteq Q \times Q$ is a set of transitions, labeled with valuations $v \in \mathbb{B}^{X_1 \cup X_2 \cup Y_1 \cup Y_2}$, and
- $F \subseteq Q$ is a set of rejecting states.

The automaton is such that it rejects a trace if it violates Φ, i.e., if rejecting states are visited infinitely often. Accordingly, it accepts a concurrent program $(P_1 \parallel P_2 \parallel \mathsf{sched}, v_0)$ if no trace in $[\![P_1 \parallel P_2 \parallel \mathsf{sched}, v_0]\!]$ violates Φ. See [18] for more background.

Let $X' = X_1' \cup X_2'$. We constrain functions f_i and μ_i with respect to an additional annotation function $\lambda : Q \times \mathbb{B}^{X'} \to \mathbb{N} \cup \{\bot\}$. In the following, let $\tau_i'(\overline{x} \circ \overline{z}, \overline{o}, \overline{y})$ denote the combined update function for the original state variables and additional memory variables, explicitly written as

$$\tau_i(\overline{x} \circ \overline{z}, \overline{o}, \overline{y} \circ f_i(\overline{x}, \overline{z}, \overline{o}, \overline{y})) \circ \mu_i(\overline{x} \circ \overline{z}, \overline{o}, \overline{y}).$$

Similar to the original bounded synthesis encoding [18], we require that

$$\lambda(q_0, v_0 \restriction_{X'}) \in \mathbb{N}.$$

If (1) $(q, (\overline{x}_1, \overline{x}_2))$ is a composed state with $\lambda(q, (\overline{x}_1, \overline{x}_2)) \in \mathbb{N}$, (2) $\overline{y}_1 \in \mathbb{B}^{Y_1}, \overline{y}_2 \in \mathbb{B}^{Y_1}$ are inputs and $q' \in Q$ is a state of the automaton such that there is a transition $(q, q') \in \Delta$ that is labeled with $(\overline{y}_1, \overline{y}_2)$, and (3) q' is a non-rejecting state of $\mathcal{U}_\Phi$, then we require

$$\lambda(q', (\tau_1'(\overline{x}_1, \overline{o}_1, \overline{y}_1), \tau_2'(\overline{x}_2, \overline{o}_2, \overline{y}_2))) \geq \lambda(q, (\overline{x}_1, \overline{x}_2)),$$

where values of $\overline{o}_1, \overline{o}_2$ are determined by values of $\overline{x}_2$ and $\overline{x}_1$, respectively (and the subset of states of one process which is observable by the other process).
Finally, if conditions (1) and (2) above hold, and q' is rejecting in $\mathcal{U}_\Phi$, we require

$$\lambda(q', (\tau_1'(\overline{x}_1, \overline{o}_1, \overline{y}_1), \tau_2'(\overline{x}_2, \overline{o}_2, \overline{y}_2))) > \lambda(q, (\overline{x}_1, \overline{x}_2)).$$

Intuitively, these constraints ensure that in no execution starting from (q_0, v_0), the automaton will visit rejecting states infinitely often. Finkbeiner and Schewe [18] have shown that these constraints are satisfiable if and only if there exist implementations of P_1, P_2 with state variables X_1, X_2 that satisfy Φ. With our additional constraints on the original τ_1, τ_2 and the integration of the f_i and μ_i as new uninterpreted functions, they are satisfiable if there exist b-bounded refinements of P_1, P_2 (based on C_1, C_2) that satisfy Φ. An SMT solver can then be used to find interpretations of the f_i and μ_i, as well as the auxiliary annotation functions that witness correctness of the refinement.

Correctness. The proposed algorithm for bounded synthesis from program sketches is correct and will eventually find a solution if it exists:

Proposition 1. *Any model of the SMT constraints will represent a refinement of the program sketches such that their composition satisfies the specification.*

Proposition 2. *There exists a model of the SMT constraints if there exist b-bounded refinements $P_1' \preceq P_1, P_2' \preceq P_2$ that satisfy the specification.*

Proof ideas for correctness can be found in an extended version [3] of this paper.

Optimization of solutions. Let $\mathsf{cost} : \mathcal{P} \times \mathcal{P} \to \mathbb{N}$ be a user-defined cost function. We can synthesize an implementation $P_1', P_2' \in \mathcal{P}$ with *maximal cost* c by adding the constraint $\mathsf{cost}(P_1', P_2') \leq c$ (and a definition of the cost function), and we can *optimize* the solution by searching for implementations with incrementally smaller cost. For instance, a cost function could count the number of memory updates in order to optimize solutions for simplicity.

5.2 SMT-based AGS

Based on the encoding from Section 5.1, this section presents an extension that solves the AGS problem. Recall that the inputs to AGS are two program sketches P_1, P_2 with $P_i = (X_i, O_i, Y_i, \tau_i)$, two sets C_1, C_2 of controllable variables with $C_i \subseteq Y_i$, two specifications Φ_1, Φ_2, and a start valuation $v_0 \in \mathbb{B}^{X \cup Y}$, where $Y = Y_1 \cup Y_2$. The goal is to obtain refinements $P_1' \preceq P_1$ and $P_2' \preceq P_2$ such that:

(i) $[\![P_1' \parallel P_2 \parallel \mathsf{sched}, v_0]\!] \subseteq (\mathsf{fair} \wedge \Phi_2 \to \Phi_1)$
(ii) $[\![P_1 \parallel P_2' \parallel \mathsf{sched}, v_0]\!] \subseteq (\mathsf{fair} \wedge \Phi_1 \to \Phi_2)$
(iii) $[\![P_1' \parallel P_2' \parallel \mathsf{sched}, v_0]\!] \subseteq (\mathsf{fair} \to \Phi_1 \wedge \Phi_2)$.

Using the approach presented above, we can encode each of the three items into a separate set of SMT constraints, using the same function symbols and variable identifiers in all three problems. In more detail, this means that we

1. encode (i), where we ask for a model of f_1 and μ_1 such that P_1' with τ_1' and P_2 with the given τ_2 satisfy the first property,
2. encode (ii), where we ask for a model of f_2 and μ_2 such that P_1 with the given τ_1 and P_2' with τ_2' satisfy the second property, and
3. encode (iii), where we ask for models of f_i and μ_i for $i \in \{1, 2\}$ such that P_1' and P_2' with τ_1' and τ_2' satisfy the third property.

Then, a solution for the conjunction of all of these constraints must be such that the resulting refinements of P_1 and P_2 satisfy all three properties simultaneously, and are thus a solution to the AGS problem. Moreover, a solution to the SMT problem exists if and only if there exists a solution to the AGS problem.

5.3 Extensions

While not covered by the definition of AGS in Section 3, we can easily extend our algorithm to the following cases:

1. If we allow the sets Z_1, Z_2 to be non-disjoint, then the synthesis algorithm can refine processes also by *adding shared variables*.
2. Also, our algorithms can easily be adapted to AGS with *more than 2 processes*, as defined in [13].

6 Experiments

We implemented[4] our approach as an extension to BoSY, the bounded synthesis backend of the parameterized synthesis tool PARTY [26]. The user defines the sketch in SMT-LIB format with a special naming scheme. The specification is given in LTL. The amount of memory is defined using an integer constant M, which is increased until a solution is found. To optimize solutions, the user can assert that some arbitrarily computed cost must be lower than some constant Opt. Our tool will find the minimal value of Opt such that the problem is still realizable. Our tool can also run cooperative co-synthesis and verify existing solutions. Due to space constraints, we can only sketch our experiments here. Details can be found in the extended version [3] of this paper.

For a simple **peer-to-peer file sharing protocol** [19], we synthesize conditions that define when a process uploads or downloads data. The specification requires that all processes download infinitely often, but a process can only download if the other one uploads. Without AGS, we obtain a brittle solution: if one process is changed to upload and download simultaneously, the other process will starve, i.e., will not download any more. The reason is that cooperative co-synthesis can produce solutions where the correctness of one process relies on a concrete realization of the other processes. With AGS, this problem does not exist. Synthesis takes only one second for this example.

Our next experiment is performed on a **double buffering** protocol, taken from [40]. There are two buffers. While one is read by P_1, the other one is written by P_2. Then, the buffers are swapped. We synthesize waiting conditions such that the two processes can never access the same buffer location simultaneously. The example is parameterized by the size N of the buffers. Table 1 lists the synthesis times for increasing N. We use bitvectors to encode the array indices, and observe that the computation time mostly depends on the bitwidth. This explains the jumps whenever N reaches the next power of two. Cooperative co-synthesis is only slightly faster than AGS on this example.

Table 1: Synthesis times [sec] for increasing N.

N	1	2	3	4	5	6	7	8	15
AGS	1	5	5	54	51	49	47	1097	877
non-AGS	1	4	4	38	35	32	31	636	447

Finally, we use our tool to **synthesize atomic sections in** a simplified version of **the i2c Linux kernel driver** in order to fix a real bug[5]. This example has been taken from [6]. We synthesize two functions f_1 and f_2 that map the program counter value of the respective process to true or false. The value true means that the process cannot be interrupted at this point in the program, i.e., the two adjacent instructions are executed atomically. We also assign costs to active atomic sections, and let our tool minimize the total costs. A meaningful solution with minimal costs is computed in 54 seconds.

7 Related Work

Reactive synthesis. Automatic synthesis of reactive programs from formal specifications, as defined by Church [14], is usually reduced either to games on finite graphs [5],

[4] Available at http://www.iaik.tugraz.at/content/research/design_verification/others.

[5] See http://kernel.opensuse.org/cgit/kernel/commit/?id=7a7d6d9c5fcd4b674da38e814cfc0724c67731b2.

or to the emptiness problem of automata over infinite trees [35]. Pnueli and Rosner [33] proposed synthesis from LTL specifications, and showed its 2EXPTIME complexity based on a doubly exponential translation of the specification into a tree automaton. We use extensions of the game-based approach (see below) to obtain new complexity results for AGS, while our implementation uses an encoding based on tree automata [18] that avoids one exponential blowup compared to the standard approaches [27].

We consider the synthesis of concurrent or distributed reactive systems with partial information, which has been shown to be undecidable in general [34], even for simple safety fragments of temporal logics [38]. Several approaches for distributed synthesis have been proposed, either by restricting the specifications to be local to each process [28], by restricting the communication graph to pipelines and similar structures [17], or by falling back to semi-decision procedures that will eventually find an implementation if one exists, but in general cannot detect unrealizability of a specification [18]. Our synthesis approach is based on the latter, and extends it with synthesis from program sketches [39], as well as the assume-guarantee paradigm [9].

Graph games. Graph games provide a mathematical foundation to study reactive synthesis problems [14,5,22]. For the traditional perfect-information setting, the complexity of solving games has been deeply studied; e.g., for reachability and safety objectives the problem is PTIME-complete [23,1]; for GR(1) the problem can be solved in polynomial time [32]; and for LTL the problem is 2EXPTIME-complete [33]. For two player partial-information games with reachability objectives, EXPTIME-completeness was established in [36], and symbolic algorithms and strategy construction procedures were studied in [8,2]. However, in the setting of multi-player partial-observation games, the problem is undecidable even for three players [31] and for safety objectives as well [11]. While most of the previous work considers only the general problem and its complexity, the complexity distinction we study for memoryless strategies, and the practical SMT-based approach to solve these games has not been studied before.

Equilibria notions in games. In the setting of two-player games for reactive synthesis, the goals of the two players are complementary (i.e., games are zero-sum). For multi-player games there are various notions of equilibria studied for graph games, such as Nash equilibria [29] for graph games that inspired notions of rational synthesis [19]; refinements of Nash equilibria such as secure equilibria [10] that inspired assume-guarantee synthesis (AGS) [9], and doomsday equilibria [7]. An alternative to Nash equilibria and its refinements are approaches based on iterated admissibility [4]. Among the various equilibria and synthesis notions, the most relevant one for reactive synthesis is AGS, which is applicable for synthesis of mutual-exclusion protocols [9] as well as for security protocols [13]. The previous work on AGS is severely restricted by perfect information, whereas we consider the problem under the more general framework of partial information, the need of which was already advocated in applications in [24].

8 Conclusion

Assume-Guarantee Synthesis (AGS) is particularly suitable for concurrent reactive systems, because none of the synthesized processes relies on the concrete realization of

the others. This feature makes a synthesized solution robust against changes in single processes. A major limitation of previous work on AGS was that it assumed perfect information about all processes, which implies that synthesized implementations may use local variables of other processes. In this paper, we resolved this shortcoming by (1) defining AGS in a partial information setting, (2) proving new complexity results for various sub-classes of the problem, (3) presenting a pragmatic synthesis algorithm based on the existing notion of bounded synthesis to solve the problem, (4) providing the first implementation of AGS, which also supports the optimization of solutions with respect to user-defined cost functions, and (5) demonstrating its usefulness by resolving sketches of several concurrent protocols. We believe our contributions can form an important step towards a mixed imperative/declarative programming paradigm for concurrent programs, where the user writes sequential code and the concurrency aspects are taken care of automatically.

In the future, we plan to work on issues such as scalability and usability of our prototype, explore applications for security protocols as mentioned in [24], and research restricted cases where the AGS problem with partial information is decidable.

References

1. C. Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Trans. on Database Systems*, 5:241–259, 1980.
2. D. Berwanger, K. Chatterjee, M. De Wulf, L. Doyen, and T. A. Henzinger. Strategy construction for parity games with imperfect information. *I&C*, 208(10):1206–1220, 2010.
3. R. Bloem, K. Chatterjee, S. Jacobs, and R. Könighofer. Assume-guarantee synthesis for concurrent reactive programs with partial information. *CoRR*, abs/1411.4604, 2014.
4. R. Brenguier, J.F. Raskin, and M. Sassolas. The complexity of admissibility in omega-regular games. In *CSL-LICS*, page 23. ACM, 2014.
5. J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the AMS*, 138:295–311, 1969.
6. P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV*, LNCS 8044, pages 951–967. Springer, 2013.
7. K. Chatterjee, L. Doyen, E. Filiot, and J-F. Raskin. Doomsday equilibria for omega-regular games. In *VMCAI*, LNCS 8318, pages 78–97. Springer, 2014.
8. K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Algorithms for omega-regular games of incomplete information. *Logical Methods in Computer Science*, 3(3:4), 2007.
9. K. Chatterjee and T. A. Henzinger. Assume-guarantee synthesis. In *TACAS*, LNCS 4424, pages 261–275. Springer, 2007.
10. K. Chatterjee, T. A. Henzinger, and M. Jurdzinski. Games with secure equilibria. *Theor. Comput. Sci.*, 365(1-2):67–82, 2006.
11. K. Chatterjee, T. A. Henzinger, J. Otop, and A. Pavlogiannis. Distributed synthesis for LTL fragments. In *FMCAD*, pages 18–25. IEEE, 2013.
12. K. Chatterjee, A. Kößler, and U. Schmid. Automated analysis of real-time scheduling using graph games. In *HSCC*, pages 163–172. ACM, 2013.
13. K. Chatterjee and V. Raman. Assume-guarantee synthesis for digital contract signing. *Formal Asp. Comput.*, 26(4):825–859, 2014.
14. A. Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*, pages 23–35, 1962.

15. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.
16. B. Finkbeiner and S. Jacobs. Lazy synthesis. In *VMCAI*, LNCS 7148, 2012.
17. B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *LICS*. IEEE, 2005.
18. B. Finkbeiner and S. Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.
19. D. Fisman, O. Kupferman, and Y. Lustig. Rational synthesis. In *TACAS*, LNCS 6015, pages 190–204. Springer, 2010.
20. S. Fortune, J.E. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theor. Comput. Sci.*, pages 111–121, 1980.
21. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, LNCS 2500. Springer, 2002.
22. Y. Gurevich and L. Harrington. Trees, automata, and games. In *STOC*, pages 60–65. ACM, 1982.
23. N. Immerman. Number of quantifiers is better than number of tape cells. *J. Comput. Syst. Sci.*, 22:384–406, 1981.
24. W. Jamroga, S. Mauw, and M. Melissen. Fairness in non-repudiation protocols. In *STM*, LNCS 7170, pages 122–139. Springer, 2011.
25. B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *J. Comput. Syst. Sci.*, 78(2):441–460, 2012.
26. A. Khalimov, S. Jacobs, and R. Bloem. PARTY parameterized synthesis of token rings. In *CAV*, LNCS 8044, pages 928–933. Springer, 2013.
27. O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *FOCS*, 2005.
28. P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. In *ICALP*, LNCS 2076, pages 396–407. Springer, 2001.
29. J.F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academny of Sciences USA*, 36:48–49, 1950.
30. C.H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
31. G. L. Peterson and J. H. Reif. Multiple-person alternation. In *FOCS*. IEEE, 1979.
32. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, LNCS 3855, pages 364–380. Springer, 2006.
33. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.
34. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE, 1990.
35. M. O. Rabin. *Automata on Infinite Objects and Churchs Problem*. American Mathematical Society, 1972.
36. J. H. Reif. The complexity of two-player games of incomplete information. *J. Comput. Syst. Sci.*, 29(2):274–301, 1984.
37. W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *JCSS*, 4(2):177 – 192, 1970.
38. S. Schewe. Distributed synthesis is simply undecidable. *IPL.*, 114(4):203–207, 2014.
39. A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
40. M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338. ACM, 2010.