Assume-Guarantee Synthesis for Concurrent Reactive Programs with Partial Information

Roderick Bloem[1], Krishnendu Chatterjee[2], Swen Jacobs[1,3], 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 very useful for concurrent systems. However, most synthesis 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 in a partial information setting. 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 [40,41,21], 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 [21,20,46], or program repair, where suitable replacements need to be synthesized for faulty program parts [30]. 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 cases: adversarial components may result in unrealizability of the system, while cooperative components may 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) [12] 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 [12] or fair-exchange protocols for digital contract signing [16].

A drawback of existing algorithms for AGS [12,16] 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 [34,21], we provide the first solution for AGS with partial information.

In this work, we extend the AGS approach for simultaneous synthesis of multiple processes with partial information restrictions, and analyze complexity and decidability of AGS for several different cases. Furthermore, we provide the first implementation of AGS, integrated into a programming model that combines the synthesis of concurrent reactive programs with ideas from program sketching. Our framework allows for a combined imperative-declarative programming style, with fine-grained, user-provided restrictions on the exchange of information between processes. Our prototype implementation also supports optimization of the synthesized program with respect to user-defined preferences, for example a small number of shared variables. We demonstrate the value of our approach on a number of small programs and protocols.

Complexity and Decidability of AGS. We use reductions of assume-guarantee synthesis problems to problems about games with three players to obtain a number of new complexity results. We distinguish the general case, where synthesized programs may contain additional variables, from the memoryless case, where no variables may be added. We provide new complexity results for these two cases in both the perfect and the partial information setting, and for specifications in different fragments of linear-time temporal logic (LTL). We show undecidability for general AGS under partial information for all fragments we consider, in particular for basic safety properties. Table 1 gives an overview of the complexity of AGS.

Algorithms for AGS. In light of the high complexity of many AGS problems, we propose a pragmatic approach, based on program sketching and synthesis with bounded resources. Inspired by the bounded synthesis approach [22], we reduce undecidable AGS problems under partial information to a sequence of decidable AGS problems with bounded memory.

To this end, we formalize how to do bounded synthesis based on a program sketch. Our synthesis algorithm uses a translation of the specification into universal co-Büchi tree automata (cf. [22]), and an encoding of the existence of a correct instantiation of the sketch into a satisfiability modulo theories (SMT) problem. We show that the approach can be extended to the AGS setting by generating a number of separate SMT problems, and searching for a solution of their conjunction.

Implementation and Evaluation. We have implemented our algorithm and provide an evaluation on a number of examples, including Peterson's mutual exclusion protocol, a P2P filesharing protocol, a double buffering protocol, and synthesis of atomic sections

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

```
turn:=F; flag1:=F; flag2:=F;

cr1:=F; wait1:=F;
do { // Process P1:
  flag1:=T;
  turn:=T;
  while(?1,1) {} //wait
  cr1:=T;
  cr1:=F; flag1:=F; wait1:=T;
  while(?1,2) {} //local work
  wait1:=F;
} while(T)

cr2:=F; wait2:=F;
do { // Process P2:
  flag2:=T;
  turn:=F;
  while(?2,1) {} //wait
  cr2:=T; //read:=?2,3
  cr2:=F; flag2:=F; wait2:=T;
  while(?2,2) {} //local work
  wait2:=F;
} while(T)
```

in a concurrent device driver. We give sketches of these protocols that leave open some decisions that are essential for correctness, and show that our AGS algorithms finds suitable solutions. Our tool also supports the optimization of the synthesized implementation with respect to different metrics like the number of memory updates or the size of atomic sections. Using this feature, we synthesize implementations that are both correct and optimal in a certain sense. Furthermore, we demonstrate how the robustness of AGS solution 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 [12], a version of Peterson's mutual exclusion protocol. More details can be found in Section 7.1.

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 P1 and P2. Variable $\texttt{flag}i$ indicates that Pi wants to enter the critical section, and $\texttt{cr}i$ that Pi 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 P1 and P2 never visit the critical section simultaneously.

Specification. The desired properties of both processes are (1) that whenever a process wants to enter the critical section, it will eventually enter it (starvation freedom), and (2) that the two processes are never in the critical section simultaneously (mutual exclusion). In Linear Temporal Logic (LTL)[1], this corresponds to the specification $\varphi_i = \mathsf{G}(\neg\texttt{cr1} \vee \neg\texttt{cr2}) \wedge \mathsf{G}(\texttt{flag}i \rightarrow \mathsf{F}\,\texttt{cr}i)$, 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 P2 may only work for the computed implementation of P1, i.e., changing P1 may break P2. For instance, the solution $?_{1,1}$ = `turn & flag2`, $?_{2,1}$ = `!turn` and $?_{i,2}$ = `F` satisfies the specification, but changing $?_{1,2}$ in P1 to `T` will make P2 starve. Note that this

[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".

is not just a hypothetical case; we got exactly this solution in our experiments (Section 7.1). . As a second option, we may assume that the processes are adversarial, i.e., P2 must work for *any* P1 and vice versa. However, under this assumption, the problem is unrealizable [12].

Success of Assume-Guarantee Synthesis (AGS) [12]. AGS fixes this dilemma by requiring that P2 must work for *any* realization of P1 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`.

Added advantage of AGS. If one process in an AGS solution is changed or extended, but still satisfies its original specification, then the other process is guaranteed to do so as well. We illustrate this feature by extending P2 with a new variable named `read`. It is updated in a yet unknown way (expressed by $?_{2,3}$) whenever P2 enters the critical section in line 26 of Listing 1. Assume 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 P2 satisfies $\varphi_2 \wedge (\mathsf{G}\,\mathsf{F}\,\neg\mathtt{read}) \wedge (\mathsf{G}\,\mathsf{F}\,\mathtt{read})$, where φ_2 is the original specification of P2. The fact that the modified process still satisfies φ_2 implies that P1 will still satisfy its original specification. We also notice that modular refinement saves overall synthesis time: our tool takes $19+55=74$ seconds to synthesize an AGS solution and refine it in a second step to get the expected solution with $?_{\mathbf{2,3}} = \neg\mathtt{read}$; direct synthesis of the refined specification for both processes requires 263 seconds.

Drawbacks of the existing [12] AGS framework. While AGS provides important improvements over classical approaches, it may still produce solutions like $?_{\mathbf{1,1}} = \mathtt{turn} \wedge \neg\mathtt{wait2}$ and $?_{\mathbf{2,1}} = \neg\mathtt{turn} \wedge \neg\mathtt{wait1}$. However, `wait2` is intended to be a *local* variable of P2, and thus invisible for P1. 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, [12] allows the user to define candidate implementations for each **?**, and let the synthesis algorithm select one of the candidates. However, 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 $f_{1,1}(\mathtt{turn},\mathtt{flag2})$ instead of $?_{1,1}$ to express that the solution may only depend on `turn` and `flag2`. Including more variables of P1 does not make sense for this example, because their value is fixed at the call site. When setting $?_{\mathbf{2,1}} = f_{1,2}(\mathtt{turn},\mathtt{flag1})$ (and $?_{\mathbf{i,2}} = f_{i,2}()$), we get the solution proposed by Peterson: $?_{\mathbf{1,1}} = \mathtt{turn} \wedge \mathtt{flag2}$ and $?_{\mathbf{2,1}} = \neg\mathtt{turn} \wedge \mathtt{flag1}$ (and $?_{\mathbf{i,2}} = \mathtt{F}$). 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: $?_{1,1} = f_{1,1}(\mathtt{flag2},\mathtt{m})$, $?_{2,1} = f_{2,1}(\mathtt{flag1},\mathtt{m})$, $?_{i,2}$ 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

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;
2  do { // Process P1:
3    flag1:=T;
4    while(!m) {} //wait
5    cr1:=T;
6    cr1:=F;  flag1:=F;  wait1:=T;
7    while(input1()) //work
8      m:=F;
9    wait1:=F;  m:=F;
10 } while(T)
```

```
21 cr2:=F;  wait2:=F;
22 do { // Process P2:
23   flag2:=T;
24   while(m) {} //wait
25   cr2:=T;
26   cr2:=F;  flag2:=F;  wait2:=T;
27   while(input2()) //work
28     m:=T;
29   wait2:=F;  m:=T;
30 } while(T)
```

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.

[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.

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.

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 $\left(\bigwedge_i \mathsf{G}\,\mathsf{F}\, L_e^i\right) \to \left(\bigwedge_j \mathsf{G}\,\mathsf{F}\, L_s^j\right)$, where the L_e^i and L_s^j are Boolean formulas over $X \cup Y$.

[3] For a definition of syntax and semantics of LTL, see e.g. [18].

– *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 defined as follows: do 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

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

Competitive co-synthesis. The *competitive co-synthesis problem* is defined as follows: do 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 defined as follows: do 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 [12] 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 $\mathsf{cost} : \mathcal{P} \times \mathcal{P} \to \mathbb{N}$ that assigns a cost to a tuple of processes. In our approach, 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. The properties of AGS allow us to replace one of the processes, say P_2: if the replacement 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, then the resulting solution 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. [16] for AGS with 3 processes.

4 Complexity and Decidability of AGS

We analyze the complexity of AGS, based on a reduction to graph-based games.

4.1 Game Graphs for Co-Synthesis

All synthesis problems defined thus far can be reduced to problems about games played on graphs with three players.

Game graphs. A 3-*player game graph* $G = ((S, E), (S_1, S_2, S_3))$ consists of a directed graph (S, E) with a finite set S of states and a set $E \subseteq S \times S$ of edges, and a partition (S_1, S_2, S_3) of the state space S into three sets. The states in S_i are player-i states, for $i \in \{1, 2, 3\}$. For a state $s \in S$, we write $E(s) = \{t \in S | (s, t) \in E\}$ for the set of successor states of s. We assume that every state has at least one outgoing edge; i.e., $E(s)$ is nonempty for all states $s \in S$.

Beginning from a start state, the three players move a token along the edges of the game graph. If the token is on a player-i state $s \in S_i$, then player i moves the token along one of the edges going out of s. The result is an infinite path in the game graph; we refer to such infinite paths as plays. Formally, a *play* is an infinite sequence $(s_0, s_1, s_2, \ldots)$ of states such that $(s_k, s_{k+1}) \in E$ for all $k \geq 0$. We write Ω for the set of plays.

Strategies. A strategy for a player is a recipe that specifies how to extend plays. Formally, a *strategy* σ_i for player i is a function $\sigma_i : S^* \cdot S_i \to S$ that, given a finite sequence of states (representing the history of the play so far) which ends in a player-i state, chooses the next state. The strategy must choose an available successor state; i.e., for all $w \in S^*$ and $s \in S_i$, if $\sigma_i(w \cdot s) = t$, then $t \in E(s)$. We write Σ_i for the set of strategies for player i.

Strategies in general require memory to remember some facts about the history of a play. An equivalent definition of strategies is as follows: Let M be a set called *memory*. A strategy $\sigma = (f, \mu)$ consists of (1) a next-state function $f : S \times M \to S$ that, given the memory and the current state, determines the successor state, and (2) a memory-update function $\mu : S \times M \to M$ that, given the memory and the current state, updates the memory.

The strategy $\sigma = (f, \mu)$ is *finite-memory* if the memory M is finite. It is *b-bounded* if $2^b \geq | M |$, and *memoryless* if M is a singleton set (i.e., $b = 0$). Memoryless strategies do not depend on the history of a play, but only on the current state. A memoryless strategy for player i can be specified as a function $f_i : S_i \to S$ such that $f_i(s) \in E(s)$ for all $s \in S_i$. Given a start state $s_0 \in S$ and three strategies $\sigma_i \in \Sigma_i$, one for each of the three players $i \in \{1, 2, 3\}$, there is a unique play, denoted $\omega(s_0, \sigma_1, \sigma_2, \sigma_3) = (s_0, s_1, s_2, \ldots)$, such that for all $k \geq 0$, if $s_k \in S_i$, then $\sigma_i(s_0, s_1, \ldots, s_k) = s_{k+1}$; this play is the outcome of the game starting at s_0 given the three strategies σ_1, σ_2, and σ_3.

In a partial information setting, players may not be able to make decisions based on the full state of the game, but only with respect to the observed state. Formally, let O be a set of observations. A *partial information strategy* with respect to an observation function $o : S \to O$ is a strategy σ with $\sigma(s_0, s_1, \ldots, s_k) = \sigma(s'_0, s'_1, \ldots, s'_k)$ whenever $o(s_i) = o(s'_i)$ for all i.

Winning. An objective $\Psi \subseteq \Omega$ is a set of plays; i.e., $\Psi \subseteq \Omega$. The following notation is derived from ATL [1]. For an objective Ψ, the set of *winning states* for player 1 in the game graph G is $\langle\langle 1\rangle\rangle_G(\Psi) = \{s \in S \mid \exists\, \sigma_1 \in \Sigma_1.\, \forall\, \sigma_2 \in \Sigma_2.\, \forall$
$\sigma_3 \in \Sigma_3.\, \omega(s, \sigma_1, \sigma_2, \sigma_3) \in \Psi\}$; a witness strategy σ_1 for player 1 for the existential quantifier is referred to as a *winning strategy*. The winning sets $\langle\langle 2\rangle\rangle_G(\Psi)$ and $\langle\langle 3\rangle\rangle_G(\Psi)$ for players 2 and 3 are defined analogously. The set of winning states for the team consisting of player 1 and player 2, playing against player 3, is $\langle\langle 1, 2\rangle\rangle_G(\Psi) = \{s \in S \mid \exists\, \sigma_1 \in \Sigma_1.\, \exists\, \sigma_2 \in \Sigma_2.\, \forall\, \sigma_3 \in \Sigma_3.\, \omega(s, \sigma_1, \sigma_2, \sigma_3) \in \Psi\}$. The winning sets $\langle\langle I\rangle\rangle_G(\Psi)$ for other teams $I \subseteq \{1, 2, 3\}$ are defined similarly.

Games based on processes and specifications. Given two processes P_1, P_2 with $P_i = (X_i, O_i, Y_i, \tau_i)$ and respective sets of controllable variables $C_i \subseteq Y_i$, we define the 3-player game graph $G = ((S, E), (S_1, S_2, S_3))$ as follows: let $S = V \times \{1, 2, 3\}$; let $S_i = V \times \{i\}$ for $i \in \{1, 2, 3\}$; and let E contain (1) all edges of the form $((v, 3), (u, i))$ for $i \in \{1, 2\}$, $v \in V$ and $u{\restriction}_{X\cup C} = v{\restriction}_{X\cup C}$, and (2) all edges of the form $((v, i), (u, 3))$ for $i \in \{1, 2\}$ and $u{\restriction}_{X_i} \in \tau_i(v{\restriction}_{X_i}, v{\restriction}_{O_i}, v{\restriction}_{Y_i})$ and $u{\restriction}_{\mathcal{X}\setminus(X_i\cup C_i)} = v{\restriction}_{\mathcal{X}\setminus(X_i\cup C_i)}$. In other words, player 1 represents process P_1, player 2 represents process P_2, and player 3 represents the environment, including the scheduler. Given a play of the form $\omega = ((v_0, 3), (v'_0, i_0), (v_1, 3), (v'_1, i_1), (v_2, 3), \ldots)$, where $i_j \in \{1, 2\}$ for all $j \geq 0$, we write $[\omega]_{1,2}$ for the sequence of valuations $(v'_0, v'_1, v'_2, \ldots)$ in ω (ignoring the intermediate valuations at player-3 states).[4]

A given specification $\Phi \subseteq V^\omega$ defines the objective $[[\Phi]] = \{\omega \in \Omega | [\omega]_{1,2} \in \Phi\}$. In this way, the specifications Φ_1 and Φ_2 for the processes P_1 and P_2 provide the objectives $\Psi_1 = [[\Phi_1]]$ and $\Psi_2 = [[\Phi_2]]$ for players 1 and 2, respectively. The objective for player 3 (the environment) is the fairness objective $\Psi_3 = \mathsf{fair}$ that both S_1 and S_2 are visited infinitely often; i.e., fair contains all plays $(s_0, s_1, s_2, \ldots) \in \Omega$ such that $s_j \in S_1$ for infinitely many $j \geq 0$, and $s_k \in S_2$ for infinitely many $k \geq 0$.

Game solutions to co-synthesis problems [12]. Based on a game graph as defined above, the cooperative co-synthesis problem for P_1, P_2, C_1, C_2, a start valuation v_0 and specifications Φ_1, Φ_2 is equivalent to finding a winning strategy for the team of players 1 and 2 from start valuation v_0, and the objective $\Psi = [[\mathsf{fair} \to \Phi_1 \wedge \Phi_2]]$. The corresponding competitive co-synthesis problem is equivalent to finding separate strategies for players $i \in \{1, 2\}$ for this game graph from start valuation v_0, and the respective objective $\Psi_i = [[\mathsf{fair} \to \Phi_i]]$.

For the AGS problem for P_1, P_2, C_1, C_2, a start valuation v_0 and specifications Φ_1, Φ_2, consider the following:

1. let $U_i = \langle\langle i\rangle\rangle_G(\mathsf{fair} \to \Psi_i)$ be the winning states for process i, based on a fair scheduler,
2. let $F_i = \langle\langle i, 3\rangle\rangle_{G{\restriction}_{U_i}}(\mathsf{fair} \wedge \Psi_i \wedge \neg\Psi_{3-i})$ be the set of states where the team of players i and 3 can win the game and force the other player to lose the game, and
3. let $W = \langle\langle 1, 2\rangle\rangle_{G{\restriction}_{S\setminus(F_1\cup F_2)}}(\mathsf{fair} \to (\Psi_1 \wedge \Psi_2))$ be the set of states where both players 1 and 2 can win the game, but not force the other to lose it, based on a fair scheduler.

[4] Note that v_j differs from v'_j only in the valuation of input variables, and v'_j differs from v_{j+1} only in the valuation of variables in $X_{i_j} \cup C_{i_j}$, controlled by process i_j.

Then the AGS problem is equivalent to finding strategies σ_1, σ_2 for players 1 and 2, respectively, such that:

1. player i wins the game with objective $(\mathsf{fair} \wedge \Psi_{3-i}) \rightarrow \Psi_i$ from all states in U_i: $\forall \sigma'_{3-i}.\ \forall \sigma_3.\ \forall s \in U_i.\ \omega(s, \sigma_i, \sigma'_{3-i}, \sigma_3) \in ((\mathsf{fair} \wedge \Psi_{3-i}) \rightarrow \Psi_i)$,
2. the team of players 1 and 2 wins the game with objective $\mathsf{fair} \rightarrow (\Psi_1 \wedge \Psi_2)$ from states $W \setminus (U_1 \cup U_2)$, and
3. $v_0 \in W$.

Formally, solving the AGS problem reduces to solving games with secure equilibria [12].

4.2 Complexity Results

Table 1 gives 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 game graph given explicitly, and the specification formula (i.e., the size of the input is the size of the explicit game graph and the length of the formula).

	Memoryless		General	
	Perfect	Partial	Perfect	Partial
Safety	P	NP-C	P	Undec
GR(1)	NP-C	NP-C	P	Undec
LTL	PSPACE-C	PSPACE-C	2EXP-C	Undec

Table 1: Complexity of Assume-Guarantee Synthesis

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.

We will explain these results in the following.

Memoryless AGS, Perfect Information. The following Theorem justifies the results in the first column of Table 1.

Theorem 1. *The complexity of memoryless AGS under perfect information is*

i) polynomial for safety properties,
ii) NP-complete for GR(1) properties, and
iii) PSPACE-complete for LTL properties.

Proof. We present the proof of the three items below.

Item i: It was shown in [12] that AGS solutions can be obtained from the solutions of games with secure equilibria. It follows from the results of [13] 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 [13] that for safety objectives, games with secure equilibria can be solved in polynomial time.

Item ii: It follows from the results of [24] 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 [39].

Item iii: 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 [18] implies PSPACE-hardness. For the upper bound, we guess a memoryless strategy (as in Item ii), and the verification problem is an LTL model checking question. Since LTL model checking is in PSPACE [18] and NPSPACE=PSPACE (by Savitch's theorem) [44,37], we obtain the desired result. □

Memoryless AGS, Partial Information. The following Theorem justifies the results in the second column of Table 1.

Theorem 2. *The complexity of memoryless AGS under partial information is*

i) NP-complete for safety properties,
ii) NP-complete for GR(1) properties, and
iii) PSPACE-complete for LTL properties.

Proof. We present the proof of the three items below.

Item i: The lower bound result was established in [15]. 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).

Item ii: The lower bound follows from Theorem 1, Item ii; and the upper bound is similar as well.

Item iii: Similar to Theorem 1, Item iii. □

General AGS, Perfect Information. The following Theorem justifies the results in the third column of Table 1.

Theorem 3. *The complexity of general AGS under perfect information is*

i) polynomial for safety properties,
ii) polynomial for GR(1) properties, and
iii) 2EXP-complete for LTL properties.

Proof. We present the proof of the three items below.

Item i: For AGS under perfect information and safety objectives, the memoryless and the general problem coincide (as mentioned in Theorem 1, Item i). The result follows from Theorem 1, Item i.

Item ii: It follows from the results of [12,13] 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 [25], the desired result follows.

Item iii: The lower bound follows from standard LTL synthesis [40]. 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 [40]. The desired result follows. □

General AGS, Partial Information. The following Theorem justifies the results in the fourth column of Table 1.

Theorem 4. *General AGS under partial information is undecidable for safety properties.*

Proof. It was shown in [38] that three-player partial-observation games are undecidable, and it was also shown that the undecidability result holds for safety objectives as well [14]. □

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 [22], 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, defines 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 reduced set of input variables of process P_i'. 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 $\Phi = (\mathsf{fair} \to \Phi_1 \wedge \Phi_2)$. 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 [22] 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 [22], 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 [22] 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.*

Proof. From our definitions of refinement and of the transition functions τ_i', it is obvious that a model will represent a refinement of the given program sketches.

Furthermore, by correctness of the annotation approach from bounded synthesis [22], any transition function that satisfies the constraints will satisfy the specification (and the combination of τ_i' is in particular a transition function). $\square$

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. Suppose such P_1', P_2' exist. By the definition of refinement, we have 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

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

The control valuations $\overline{c}$ for different valuations $\overline{x}, \overline{o}, \overline{y}$ of the other variables give us a model of the function f_i that computes the controllable variables. In a similar way, the computation of memory valuations for different valuations of the other variables gives us a model of the function μ_i. $\square$

Optimization of solutions. Let $\mathsf{cost} : \mathcal{P} \times \mathcal{P} \to \mathbb{N}$ be a user-defined const function. We can synthesize an implementation $P_1', P_2' \in \mathcal{P}$ with *maximal cost* b by adding the constraint $\mathsf{cost}(P_1', P_2') \leq b$ (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) $[\![P_1' \parallel P_2 \parallel \mathsf{sched}, v_0]\!] \subseteq (\mathsf{fair} \wedge \Phi_2 \rightarrow \Phi_1)$
(ii) $[\![P_1 \parallel P_2' \parallel \mathsf{sched}, v_0]\!] \subseteq (\mathsf{fair} \wedge \Phi_1 \rightarrow \Phi_2)$
(iii) $[\![P_1' \parallel P_2' \parallel \mathsf{sched}, v_0]\!] \subseteq (\mathsf{fair} \rightarrow \Phi_1 \wedge \Phi_2)$.

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 [16].

6 Implementation

We have implemented our AGS approach with partial information as an extension to BoSY, the bounded synthesis backend of parameterized synthesis tool PARTY [31]. It uses LTL3BA [2] to transform specifications into automata, and Z3 [19] as SMT solver. Our extension is available for download[5].

[5] `http://www.student.tugraz.at/robert.koenighofer/tacas15_AG.zip`

Input. Our tool takes three input files: a program sketch and one specification file for each process. The sketch is defined directly in SMT-LIBv2 [3] format, the specifications are given in LTL, using the Acacia [6] syntax.

The sketch defines data types for the state space $\mathbb{B}^X$, the uncontrollable input space $\mathbb{B}^{Y'}$, the controllable input space $\mathbb{B}^C$, and (optional) memory $\mathbb{B}^Z$, along with the initial valuation v_0 of all variables.[6] For each Boolean signal s that appears in the specification, the sketch defines a labeling function $s : \mathbb{B}^X \times \mathbb{B}^Y \to \mathbb{B}$, which is by default just the value of a state or input variable. For signals of the specification that are not directly available as state- or input variables, the labeling function needs to be explicitly defined.

In our experiments, we mostly use bitvectors of appropriate length to define state space, inputs, and memory. However, our tool also supports the definition of user-defined data types such as tuples of enumeration types, which may be more convenient for other applications. Our tool uses a special integer constant M to refer to the number of memory variables per process, and increases M until a solution is found. All memory variables are global by default. Partial information is modeled by restricting the set of variables on which the functions that control the strategy or update the memory can depend. This fine-grained definition of partial information increases the flexibility of our tool.

By default, the sketch defines the (global) transition function τ as the parallel composition of the transition functions τ_1 and τ_2 of the processes, but sometimes defining the combined transition function directly is easier. Finally, the sketch declares the functions that should be synthesized: two control functions f_i, and two memory update functions μ_i. The user can specify each of these functions compositionally, with multiple sub-functions that control disjoint subsets of C_i or Z_i, respectively. For each sub-function, observable variables can be defined individually, allowing for a very fine-grained use of partial information.

Optimization of solutions. In order to facilitate the optimization of solutions, the user can assert that some arbitrarily computed cost has to be lower than some special constant Opt in the sketch file. Our tool will find the minimal value of Opt, within a user-defined interval, such that the problem is still realizable. At the moment, this search is implemented in a straightforward way: Opt is decreased (increased) by 1 as long the problem is realizable. More sophisticated search strategies like binary search, learning from failed attempts, or using incremental solving are possible but not yet implemented. With a solver for MAX-SMT problems, this search could also be entrusted to the solver.

Other applications. Our tool can also complete sketches with cooperative co-synthesis (see Section 3). Furthermore, we can use our tool as a model checker for solutions by completely defining the functions f_i and μ_i in the sketch instead of just declaring them.

The fact that our tool takes as input an SMT-LIBv2 formulation of the synthesis problem makes it very flexible: By defining the transition relation appropriately, it can also be used for synthesis of entire systems from scratch, or synthesis of atomic sections in concurrent programs. Furthermore, additional requirements on the solution can be defined easily with additional SMT constraints. The obvious downside is limited us-

[6] Wlog., we assume that memory variables are initialized to a default value, e.g. false for Boolean variables.

Listing 3: Synthesis result for the sketch in Listing 1 without AGS.

```
0                              turn:=F; flag1:=F; flag2:=F;
1 cr1:=F; wait1:=F;
2 do { // Process P1:
3    flag1:=T;
4    turn:=T;
5    while(turn & flag2) {} //wait
6    cr1:=T;
7    cr1:=F; flag1:=F; wait1:=T;
8    while(F) {} //local work
9    wait1:=F;
10 } while(T)
```

```
21 cr2:=F; wait2:=F;
22 do { // Process P2:
23    flag2:=T;
24    turn:=F;
25    while(!turn) {} //better: & flag1
26    cr2:=T;
27    cr2:=F; flag2:=F; wait2:=T;
28    while(F) {} //local work
29    wait2:=F;
30 } while(T)
```

ability, since defining the program semantics in SMT-LIBv2 format is not always easy. In future work, we want to implement a front-end to define simple sketching problems in a subset of C in order to increase the usability.

7 Experiments

All experiments in this section were performed on an ordinary notebook (Intel i5-3320M CPU@2.6 GHz, 8 GB RAM, 64-bit Linux), using one CPU core and a memory limit of 1 GB, which was never reached.

7.1 Peterson's Mutual Exclusion Protocol

This example has already been used as motivation in Section 2. In this section, we give additional insights, experiments and performance measures.

In our model of this program, we use a bitvector of size 7 to represent states: Each process has a program counter of 3 bits (assignments written in the same line in Listing 1 are executed simultaneously), and one bit is used to model the variable `turn`. The current value of all other variables can be computed from the respective program counter value. Hence, they are modeled with state labels.

AGS without partial information. When using AGS without any restrictions on variable dependencies, our tool takes 26 seconds to find a solution. However, it is too complicated to be shown here, let alone understand it. The question marks are implemented as what appears to be arbitrary functions over all variables, including program counter bits from the other process. This solution is overly complicated and thus clearly undesirable. This motivates our partial information extension to AGS.

Cooperative co-synthesis. In our next experiment, we therefore restrict the observable information for resolving the question marks by setting $\mathbf{?}_{1,1} = f_{1,1}(\texttt{turn}, \texttt{flag2})$, $\mathbf{?}_{2,1} = f_{2,1}(\texttt{turn}, \texttt{flag1})$, and $\mathbf{?}_{i,2} = f_{i,2}()$. Furthermore, we disable AGS (i.e., use cooperative co-synthesis) and do not allow extra memory. Our tool takes 7 seconds to find the solution shown in Listing 3. The problem with this solution is that P2 relies on the concrete realization of P1. If we would later modify the condition in line 8 (i.e., $\mathbf{?}_{1,2}$) to true, then P2 would starve while waiting for P1 to set `turn`.

AGS with partial information. AGS prevents such dependencies on the concrete realization of other processes, thereby making the solution robust against a posteriori

Listing 4: Synthesis result for the sketch in Listing 1 with AGS and memory, but without optimization for simplicity.

```
0                   flag1:=F; flag2:=F; m:=F;
```

```
1  cr1:=F; wait1:=F;
2  do { // Process P1:
3    flag1:=T; m:=F;
4    while(flag2 & !m) {}
5    cr1:=T;
6    cr1:=F; flag1:=F; wait1:=T;
7    while(F) //wait
8      m:=F;
9    wait1:=F; m:=F;
10 } while(T)
```

```
21 cr2:=F; wait2:=F;
22 do { // Process P2:
23   flag2:=T; m := !m;
24   while(m) {} //wait to enter
25   cr2:=T;
26   cr2:=F; flag2:=F; wait2:=T; m:=F;
27   while(F) //wait
28     m:=F;
29   wait2:=F; m:=F;
30 } while(T)
```

changes of single processes. Indeed, when running our tool *with* AGS, we get `!turn & flag1` in line 25, which resolves the problem. The execution time increases from 7 to 19 seconds, which is acceptable.

Introducing memory. So far, we assumed that the synchronization variables are already present. However, by introducing additional memory variables, our synthesis approach can also invent them. In our next experiment, we remove `turn`, allow some memory m to be updated based on the program counter (of the currently scheduled process) and the old memory, and set $\mathbf{?}_{1,1}= f_{1,1}(\texttt{m}, \texttt{flag2})$, and $\mathbf{?}_{2,1}= f_{2,1}(\texttt{m}, \texttt{flag1})$. We get the solution depicted in Listing 4 within 19 seconds. The synthesis tool re-invents `turn`, but the solution is complicated. We had to construct a graph summarizing all runs to certify that the specification holds. This motivates our optimization feature, which can be used to obtain simple solutions.

Optimization. Next, we therefore add to the SMT formulation of the synthesis problem constraints that count the updates of m in an integer variable c, and also add the constraint $c <$ Opt. Now, we let the synthesis tool find a minimum value for Opt such that the problem is still realizable. Doing this, the tool will find an overly simplistic solution: it sets the waiting condition $\mathbf{?}_{i,2}$ to true, which means that the synchronization needs to work only once. When we consider the waiting condition to be an input, we get the solution shown in Listing 2, which has already been discussed in Section 2.

Refinement. In Section 2, we already discussed the refinement of the basic AGS solution with an additional variable `read`. Next, we refine the version with memory (Listing 2) in the same way. By setting $\mathbf{?}_{2,3}= f_{2,3}(\texttt{read})$, our tool finds the expected solution $f_{2,3}(\texttt{read}) = \neg\texttt{read}$ of toggling `read` whenever the critical section is entered within 58 seconds. Again, the modular refinement saved synthesis time. Instead of $74 + 58 = 132$ seconds for synthesizing an AGS solution and refining it later, direct synthesis of the refined specification for both processes simultaneously requires 266 seconds.

7.2 Peer-To-Peer Filesharing

Sketch. This example has been taken from [23] with slight modifications. Two processes P1 and P2 use a peer-to-peer protocol to share files. In each step, process Pi can decide whether it wants to upload (by setting the variable ui) or download (by setting di). A process can only download if the other one uploads. This is formalized by the sketch in Listing 5.

Listing 5: Sketch of a filesharing protocol.

```
   u1:=T; d1:=F; u2:=F; d2:=F;
do{//process P1            do{//process P2
 u1 := ?1,1                 u2 := ?2,1
 d1 := u2 & ?1,2            d2 := u1 & ?2,2
} while(T)                 } while(T)
```

Specification. Process $P1$ is specified by $(\mathsf{G}\,\mathsf{F}\,\mathtt{d1}) \wedge (\mathsf{G}\,\mathsf{F}(\mathtt{u1} \wedge \text{scheduled}(P1)))$ and $P2$ is specified by $(\mathsf{G}\,\mathsf{F}\,\mathtt{d2}) \wedge (\mathsf{G}\,\mathsf{F}(\mathtt{u2} \wedge \text{scheduled}(P2)))$. The first conjunct of each specification expresses the goal of downloading infinitely often. The second gives the other process the chance to do the same.

Results. We set $?_{1,j} = f_{1,j}(\mathtt{d2}, \mathtt{u2})$ and $?_{2,j} = f_{2,j}(\mathtt{d1}, \mathtt{u1})$, i.e., $P1$ makes its upload/download decisions based on the status of $P2$ and vice versa. Without AGS, we could[7] get the solution in Listing 6. Figure 1 summarizes all executions that are possible in this implementation. Edges are labeled with scheduling decisions, and states are labeled by the values of u1, d1, u2, and d2 in this order. Given that the scheduler is fair, all depicted states will be visited infinitely

Listing 6: Solution without AGS.

```
   u1:=T; d1:=F; u2:=F; d2:=F;
do{// process P1           do{// process P2
 u1 := (d2==u2);            u2 := (u1==d1)
 d1 := u2 & !d2             d2 := u1 & !d1
} while(T)                 } while(T)
```

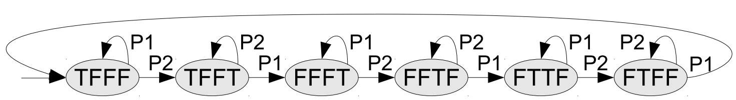

Fig. 1: Run-graph summarizing all executions of Listing 6.

often, so the specification of both processes is fulfilled. However, the correctness of this solution depends on the fact that no process ever uploads and downloads simultaneously. If one process does, the other one gets stuck in a state where it uploads but never downloads. As a concrete example, consider an alternative implementation of P2 with $?_{2,j} =$ true. That is, P2 always uploads and downloads at the same time. The entire system will get stuck in state TFTT, so the change in P2 makes P1 starve, although the specification of P2 is still satisfied. Using our approach of AGS, we can be sure that specification-preserving changes to one process cannot affect the correctness of the other. Our tool computes an AGS solution within one second for this example.

[7] Without AGS, our tool could have produced this solution, but it actually produces a different one. The produced solution does not satisfy the AGS requirements either, but in a way that is more difficult to explain.

Listing 7: Sketch of a double buffering application.

```
0                         fill:=T; render:=F;
1 i:=0; wait1:=F;
2 do { // process P1
3  while(i < N) {
4   buf[fill][i] := read();
5   i := i + 1;
6  }
7  fill:=!fill; wait1:=T;
8  while(?1)//Sol.: fill == render
9   { } // busy wait
10 i:=0; wait1:=F;
11 } while(T)
```

```
21 j:=0; wait2:=F;
22 do { // process P2
23  while(j < N) {
24   write(buf[render][j])
25   j := j + 1;
26  }
27  render:=!render; wait2:=T;
28  while(?2)//Sol.: fill != render | !wait1
29   { } // busy wait
30  j:=0; wait2:=F;
31 } while(T)
```

7.3 Double-Buffering

Sketch. The example in Listing 7 is taken from [49] with slight adaptions. It models a variant of the producer-consumer problem. There are two buffers, `buf[0]` and `buf[1]`. While process P1 writes to `buf[0]`, P2 reads from `buf[1]`. Then, the buffers are swapped. Such double-buffering is used in computer graphics and device drivers. We want to synthesize a rendezvous so that the two processes can never access the same buffer location simultaneously. Hence, our (initial) specification for both processes is $\mathsf{G}(\neg\texttt{P1w} \vee \neg\texttt{P2r} \vee \texttt{fill} \neq \texttt{render} \vee i \neq j)$, where `P1w` indicates that P1 is in line 4, and `P2r` indicates that P2 is in line 24.

Results. Our synthesis tool satisfies this specification with $\textbf{?}_i\!=$ true, so we add the progress properties $\mathsf{G\,F}(\texttt{P1w})$ and $\mathsf{G\,F}(\texttt{P2r})$ to get more meaningful solutions. With $\textbf{?}_i\!= f_i(\texttt{fill}, \texttt{render})$, the tool reports unrealizability (without memory). The solution of waiting while `fill=render` does not work because P2 could be stuck in line 28 without being scheduled until P1 flips `fill` again, which produces a deadlock. But intuitively, there should exist a solution utilizing the equality `fill=render`. Next, we therefore set $\textbf{?}_1\!= f_1(\texttt{fill=render}, \texttt{wait2})$ and $\textbf{?}_2\!= f_2(\texttt{fill=render}, \texttt{wait1})$. This allows processes to observe whether the other one is waiting. For $N = 1$, we get the solution printed in comments in Listing 7. Essentially, the two processes take turns: by having opposite waiting conditions (`fill=render` vs. $\texttt{fill}\neq\texttt{render}$) one waits while the other works. The additional disjunct `!wait1` in P2 is only useful in the first iteration: if P2 finishes first, it waits although $\texttt{fill}\neq\texttt{render}$.

Performance. Table 2 lists the synthesis times for resolving the sketch of Listing 7 with increasing N. We use bitvectors for encoding the counters i and j, and observe that the computation time mostly depends on the bit-width. 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 2: Synthesis times [sec] for Listing 7.

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

7.4 Synthesis of Atomic Sections in a Driver

Program. This example is taken from [9] (called `ex5` there), and is a simplified version of a bug in the i2c driver of the Linux kernel[8]. The code is shown in Listing 8. Process P1 opens sessions and P2 closes them. The variable `Open` counts the currently opened sessions. If there are open sessions, `On` is set to true, otherwise to false. Due to a race condition, it can happen that $\texttt{Open} \neq 0$, but $\texttt{On} = \textsf{false}$.[9] We will now use our engine to synthesize atomic sections so that this problem cannot occur.

Listing 8: Bug in i2c driver (simplified).

```
0               Open:=0; On:=F;
1  do{ //process P1
2    if(Open < MAX){
3      if(Open == 0)
4        On := T;
5      Open++;
6    }
7  } while(T)

21 do{ //process P2
22   if(Open > 0){
23     Open--;
24     if(Open == 0)
25       On := F;
26   }
27 } while(T)
```

Modeling. We search for two functions f_1 and f_2 that map the program counter value of the respective process to true or false. If a program counter value is mapped to true, then this means that the process cannot be interrupted at this point in the program, but immediately continues to execute the next instruction. That is, the two adjacent instructions are executed atomically. (Each line is considered an instruction.) In the SMT input file to our synthesis tool, this is modeled by making process P1 do nothing if it is scheduled but $f_2(pc2)$ is true, and vice versa. This way, we do not have to change the scheduler to take atomic sections into account, but rather ignore "wrong" scheduling decisions in our transition relation, which has the same effect under a fair scheduler.

Specification. Using $\Phi = \mathsf{G}((\texttt{Open} = 0) \vee \texttt{On})$ as the sole specification for both processes is not ideal: one process could make the other starve by building an atomic loop (e.g, by making all statements atomic). This enforces the specification, but is not desirable. Hence, we specify process P1 by $\Phi \wedge \mathsf{G}(\mathsf{F}(\text{scheduled}(P2) \wedge \neg f_1(pc1)))$ and P2 by $\Phi \wedge \mathsf{G}(\mathsf{F}(\text{scheduled}(P1) \wedge \neg f_2(pc2)))$. This way, both processes allow the other one to move infinitely often.

Results. For performance reasons, we prefer solutions with a low number of atomic sections. Hence, we assign costs to active atomic sections, and let our tool minimize the total costs. As a result, we get an atomic section between line 4 and 5, and another one between line 24 and 25. This renders all updates of the variable `On` atomic with the relevant accesses of `Open`, and thus fixes the race condition. Both AGS and cooperative co-synthesis produce the same solution for this example within 54 and 35 seconds.

8 Related Work

Reactive synthesis. Automatic synthesis of reactive programs from formal specifications, as defined by Church [17], is usually reduced either to games on finite graphs [8], or to the emptiness problem of automata over infinite trees [42]. Pnueli and Rosner [40] proposed synthesis from LTL specifications, and showed its 2EXPTIME complexity

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

[9] by executing the lines 2, 3, 4, 5, 22, 23, 24, 2, 3, 4, 5, 25 in a row.

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 [22] that avoids one exponential blowup compared to the standard approaches [33].

We consider the synthesis of concurrent or distributed reactive systems with partial information, which has been shown to be undecidable in general [41], even for simple safety fragments of temporal logics [45]. Several approaches for distributed synthesis have been proposed, either by restricting the specifications to be local to each process [34], by restricting the communication graph to pipelines and similar structures [21], 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 [22]. Our synthesis approach is based on the latter, and extends it with synthesis from program sketches [46], as well as the assume-guarantee paradigm [12].

Graph games. Graph games provide a mathematical foundation to study the reactive synthesis problem [17,8,27]. 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 [28,4]; for GR(1) the problem can be solved in polynomial time [39]; and for LTL the problem is 2EXPTIME-complete [40]. For two player partial-information games with reachability objectives, EXPTIME-completeness was established in [43], and symbolic algorithms and strategy construction procedures were studied in [11,5]. However, in the setting of multi-player partial-observation games, the problem is undecidable even for three players [38] and for safety objectives as well [14]. 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.

Various 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 [36] for graph games that inspired notions of rational synthesis [23]; refinements of Nash equilibria such as secure equilibria [13] that inspired assume-guarantee synthesis (AGS) [12], and doomsday equilibria [10]. An alternative to Nash equilibria and its refinements are approaches based on iterated admissibility [7]. 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 [12] as well as for security protocols [16]. 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 [29]).

Synthesis of program fragments, sketching. For functional programs, where the specification is a relation between a single pair of inputs and outputs that can be represented as a first-order logic formula, early works [50,26,35] were based on extensions of first-order theorem provers with induction and proof analysis. Recent methods leverage the power of decision procedures to obtain completeness even when reasoning about infinite data types [32], as well as techniques that limit the control structure of the synthesized program by bounding the resources of the program [48].

A special form of the latter approach is *program sketching* [47,46], where the control structure of the program is given, and values for a fixed number of unknown variables are determined by search techniques. Our approach is inspired by program sketching, in that we use sketches to limit the control structure of synthesized programs. We go beyond standard program sketching in that our programs are reactive, and in general can use an unbounded amount of memory in addition to the program variables in the sketch. Moreover, the search for suitable valuations of variables in sketching is usually implemented as a counterexample-guided inductive synthesis (CEGIS) loop, whereas we use an automata-based approach that encodes the existence of a solution into a single SMT problem. In synthesis of reactive systems, synthesis from *partial designs* allows to start with a distributed system where some components are already implemented [21,22], and an approach similar to CEGIS has been proposed as *lazy synthesis* [20].

9 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 [29], and research restricted cases where the AGS problem with partial information is decidable.

References

1. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.
2. T. Babiak, M. Kretínský, V. Rehák, and J. Strejcek. LTL to Büchi automata translation: Fast and more deterministic. In *TACAS*, LNCS 7214, pages 95–109. Springer, 2012.
3. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. In *SMT*, 2010.
4. C. Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Trans. on Database Systems*, 5:241–259, 1980.
5. 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.

6. A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J.-F. Raskin. Acacia+, a tool for LTL synthesis. In *CAV*, LNCS 7358, pages 652–657. Springer, 2012.
7. R. Brenguier, J.F. Raskin, and M. Sassolas. The complexity of admissibility in omega-regular games. In *CSL-LICS*, page 23. ACM, 2014.
8. J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the AMS*, 138:295–311, 1969.
9. 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.
10. 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.
11. 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.
12. K. Chatterjee and T. A. Henzinger. Assume-guarantee synthesis. In *TACAS*, LNCS 4424, pages 261–275. Springer, 2007.
13. K. Chatterjee, T. A. Henzinger, and M. Jurdzinski. Games with secure equilibria. *Theor. Comput. Sci.*, 365(1-2):67–82, 2006.
14. K. Chatterjee, T. A. Henzinger, J. Otop, and A. Pavlogiannis. Distributed synthesis for LTL fragments. In *FMCAD*, pages 18–25. IEEE, 2013.
15. K. Chatterjee, A. Kößler, and U. Schmid. Automated analysis of real-time scheduling using graph games. In *HSCC*, pages 163–172. ACM, 2013.
16. K. Chatterjee and V. Raman. Assume-guarantee synthesis for digital contract signing. *Formal Asp. Comput.*, 26(4):825–859, 2014.
17. A. Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*, pages 23–35, 1962.
18. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.
19. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, LNCS 4963, pages 337–340. Springer, 2008.
20. B. Finkbeiner and S. Jacobs. Lazy synthesis. In *VMCAI*, LNCS 7148, 2012.
21. B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *LICS*. IEEE, 2005.
22. B. Finkbeiner and S. Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.
23. D. Fisman, O. Kupferman, and Y. Lustig. Rational synthesis. In *TACAS*, LNCS 6015, pages 190–204. Springer, 2010.
24. S. Fortune, J.E. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theor. Comput. Sci.*, pages 111–121, 1980.
25. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, LNCS 2500. Springer, 2002.
26. C. Cordell Green. Application of theorem proving to problem solving. In *IJCAI*, pages 219–240. William Kaufmann, 1969.
27. Y. Gurevich and L. Harrington. Trees, automata, and games. In *STOC*, pages 60–65. ACM, 1982.
28. N. Immerman. Number of quantifiers is better than number of tape cells. *J. Comput. Syst. Sci.*, 22:384–406, 1981.
29. W. Jamroga, S. Mauw, and M. Melissen. Fairness in non-repudiation protocols. In *STM*, LNCS 7170, pages 122–139. Springer, 2011.
30. B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *J. Comput. Syst. Sci.*, 78(2):441–460, 2012.
31. A. Khalimov, S. Jacobs, and R. Bloem. Party parameterized synthesis of token rings. In *CAV*, LNCS 8044, pages 928–933. Springer, 2013.
32. Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Functional synthesis for linear arithmetic and sets. *STTT*, 15(5-6):455–474, 2013.

33. O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *FOCS*, 2005.
34. P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. In *ICALP*, LNCS 2076, pages 396–407. Springer, 2001.
35. Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.
36. J.F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academny of Sciences USA*, 36:48–49, 1950.
37. C.H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
38. G. L. Peterson and J. H. Reif. Multiple-person alternation. In *FOCS*. IEEE, 1979.
39. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, LNCS 3855, pages 364–380. Springer, 2006.
40. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.
41. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE, 1990.
42. M. O. Rabin. *Automata on Infinite Objects and Churchs Problem*. American Mathematical Society, 1972.
43. J. H. Reif. The complexity of two-player games of incomplete information. *J. Comput. Syst. Sci.*, 29(2):274–301, 1984.
44. W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *JCSS*, 4(2):177 – 192, 1970.
45. S. Schewe. Distributed synthesis is simply undecidable. *IPL.*, 114(4):203–207, 2014.
46. A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
47. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.
48. Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.
49. M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338. ACM, 2010.
50. Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. In *IJCAI*, pages 241–252. William Kaufmann, 1969.