Bounded Synthesis of Synchronized Distributed Models from Lightweight Specifications

Pablo Francisco Castro
Universidad Nacional de Río Cuarto - CONICET
Rio Cuarto, Córdoba, Argentina
pcastro@dc.exa.unrc.edu.ar

Luciano Putruele
Universidad Nacional de Río Cuarto
Rio Cuarto, Argentina
lputruele@dc.exa.unrc.edu.ar

Renzo Degiovanni
Luxembourg Institute of Science and Technology
Luxembourg, Luxembourg
renzo.degiovanni@list.lu

Nazareno Aguirre
Universidad Nacional de Río Cuarto - CONICET
Rio Cuarto, Argentina
naguirre@dc.exa.unrc.edu.ar

Abstract

We present a novel approach to synthesizing synchronized process designs from lightweight formal specifications. This method takes as input a collection of process specifications expressed in terms of pre/post-conditions, together with a global constraint, expressed in linear-time temporal logic, to be fulfilled by the interaction of these processes. The technique uses the *Alloy Analyzer* to enumerate potential implementations of the specified components up to a given bound, the NuSMV model checker to verify whether their concurrent composition satisfies the global property, and produces executable implementations of the processes in the style of the Promela language. Due to the exponential number of candidate implementations, the synthesis algorithm employs a two-step process to efficiently prune the solution space. During the *exploration* phase, our approach first collects a *batch of counterexamples* that are later used during the *exploitation* phase to speed up the search.

CCS Concepts

• **Software and its engineering → Automatic programming**.

Keywords

Program Synthesis, Formal Methods, Program Verification

ACM Reference Format:
Pablo Francisco Castro, Luciano Putruele, Renzo Degiovanni, and Nazareno Aguirre. 2026. Bounded Synthesis of Synchronized Distributed Models from Lightweight Specifications. In *IEEE/ACM 14th International Conference on Formal Methods in Software Engineering (FormaliSE '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3793656.3793696

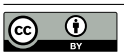
1 Introduction

Program synthesis [12, 20, 25] enables the automated construction of programs from specifications. Ideally, a developer only provides a formal description of the desired program behavior, from which a correct implementation is automatically generated. This approach allows developers to focus on high-level specifications rather than low-level implementation details. Recent advances in synthesis techniques and computing power have enabled the application of program synthesis to increasingly complex case studies (see, e.g., [7]).

In this paper, we focus on the synthesis of high-level descriptions of distributed systems, such as communication protocols and multi-agent systems. These systems have become crucial in modern software applications, including collaborative tools, communication networks, and cryptocurrencies. Moreover, their correct implementation is generally rather complex and time-consuming, which often results in subtle programming errors. This is particularly the case when synchronization primitives (e.g., semaphores, locks, monitors) are used to manage the access to shared resources. The incorrect use of these mechanisms often leads to challenging issues that are typically difficult to identify and fix, such as race conditions and deadlocks. Consequently, distributed algorithms that use synchronization mechanisms constitute a compelling target for synthesis approaches.

Existing techniques for program synthesis include game-theoretic methods for temporal-logic specifications [24], synthesis from input–output examples [18], inductive programming [1], and theorem-proving approaches [28]. These approaches have been successfully applied in various domains. In contrast, synthesis techniques for distributed algorithms are less common. As mentioned above, this setting introduces additional challenges, including handling multiple interacting processes, complying with specific communication architectures, and correctly instrumenting coordination via synchronization mechanisms. In fact, the problem of synthesizing distributed code from specifications may be undecidable in many settings [26].

The (bounded) synthesis technique presented in this paper targets asynchronous distributed systems that communicate through shared memory and synchronize via locks. In this setting, a system specification consists of a finite collection of process (or component) specifications, each of which is given in terms of actions defined via pre/post-conditions, along with local constraints expressing required properties of action executions. Additionally, the specification includes a global temporal property that the overall system behavior must satisfy. Local constraints are expressed in Alloy's relational logic [17], a first-order logic with transitive closure (necessary for expressing certain constraints, notably reachability),



FormaliSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2478-7/26/04
https://doi.org/10.1145/3793656.3793696

whereas global constraints are specified in linear-time logic (LTL). If successful, the technique produces implementations of the components that use their corresponding local actions and lock-based synchronizations, ensuring that both local and global requirements are met.

More specifically, the approach encodes component specifications as Alloy models [17], so that the models' satisfying instances correspond to locally correct implementations of the respective processes. Thus, we indirectly use Boolean satisfiability to enumerate (bounded) locally valid implementations of processes. Once locally valid implementations of processes are obtained, their composition is verified against the *global* property to determine whether the current local candidates yield a correct distributed solution. Due to efficiency reasons, the latter step is performed using a symbolic model checker. The overall search proceeds lexicographically, employing backtracking when all possible instances of a component specification have been exhausted.

To address the exponential growth in the number of possible component implementations and their potential interactions, our approach leverages batches of counterexamples to accelerate the synthesis process. This is implemented in two phases: an *exploration phase* that collects instances from the solver and generates counterexamples via model checking, and an *exploitation phase* that refines the component specifications based on these counterexamples. If no solutions are found, the process is repeated with the refined specifications. The efficiency and completeness of the algorithm depend on the chosen batch configuration, which we explore in Section 7 and discuss the corresponding trade-offs.

The paper is structured as follows. In Section 2, we introduce preliminary concepts. Section 3 illustrates our approach through a motivating example. Sections 4 and 5 present the formal framework we use to write system specifications. Sections 6 and 7 describe the main algorithm and discuss some experimental results. Finally, Sections 8 and 9 discuss related work and present some final remarks.

2 Preliminaries

Notation. Throughout this paper, we use the following notation, $(x_0, \ldots, x_n)\!\uparrow\! i$ denotes the i-th projection of tuple $(x_0, \ldots, x_n)$. $[0, k]$ denotes the set $\{0, \ldots, k\}$, $\prod_{i \in I} S_i$ refers to the Cartesian product of sets S_i, and $\coprod_{i \in I} S_i$ denotes the coproduct of sets S_i.

Labeled Transition Systems. A well-established approach to providing semantics for reactive and distributed systems is the use of *Labeled Transition Systems* (LTSs). An LTS is a tuple $\langle S, Act, \rightarrow, I, AP, L\rangle$ where: S is a (finite) non-empty set of *states*, Act is a (finite) set of *actions*, $\rightarrow \subseteq S \times Act \times S$ is a labeled transition relation, $I \subseteq S$ is a set of *initial states*, AP is a (finite) set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labeling function. Intuitively, S serves as an abstraction of the system's states, L indicates which atomic properties hold in each state, Act identifies the system's actions, and $\rightarrow$ captures the behavior of actions. For convenience, we write $s \xrightarrow{a} s'$ or $a(s, s')$ instead of $\langle s, a, s'\rangle \in \rightarrow$. Similarly, we write $s \rightarrow s'$ or $Post(s, s')$ if there is an $a \in Act$ such that $s \xrightarrow{a} s'$. The reflexive-transitive closure of the relation $\rightarrow$ helps to capture system executions, and is written $\rightarrow^*$. A path over an LTS is a sequence of alternating states and actions $\pi = s_0, a_0, s_1, a_1 \ldots$ such that $\forall i : s_i \xrightarrow{a_i} s_{i+1}$. We say that a path is an *execution* (or *trace*) if it is maximal. Without loss of generality, we assume that each state has at least one successor. When only states are relevant, we refer to executions and paths as sequences of states. We extend L to paths as follows: $L(\pi) = L(s_0, a_0, s_1, a_1, s_2, \ldots) = L(s_0), L(s_1), L(s_2), \ldots$. When convenient, we write $x(s)$ instead of $x \in L(s)$. From now on, $Path(T)$ denotes the set of paths of T, $Traces(T)$ denotes its set of executions.

(Extended) First-Order Logic over LTSs. Properties over LTSs will be expressed using first-order formulas. The logic is essentially the relational logic underlying Alloy [17]. For clarity, we will use standard first-order logic extended with transitive closure. Given actions Act and a set AP of atomic propositions, the basic vocabulary consists of binary predicates $\{a \mid a \in Act\} \cup \{Post\}$, unary predicates $\{p \mid p \in AP\} \cup \{I\}$; the logical operators are the usual Boolean connectives and first-order quantifiers with variables ranging over states, when useful we also consider constants denoting fixed states. We consider the reflexive-transitive closure operator, written a^* (for any binary predicate a). Reflexive-transitive closure is necessary to express reachability constraints. Given an LTS T, a formula ϕ, and a interpretation of variables to states $\mathcal{I} : AP \rightarrow S$, we can define the relation $T, \mathcal{I} \vDash \phi$ as follows:

- $T, \mathcal{I} \vDash p_i(x)$ iff $p_i \in L(\mathcal{I}(x))$, for $p_i \in AP$,
- $T, \mathcal{I} \vDash r(x, y)$ iff $r(\mathcal{I}(x), \mathcal{I}(y))$, for $r \in Act \cup \{Post\}$,
- $T, \mathcal{I} \vDash r^*(x, y)$ iff $r^*(\mathcal{I}(x), \mathcal{I}(y))$, for $a \in Act \cup \{Post\}$,
- $T, \mathcal{I} \vDash \phi \wedge \psi$ iff $T, \mathcal{I} \vDash \phi$ and $T, \mathcal{I} \vDash \psi$,
- $T, \mathcal{I} \vDash \neg\psi$ iff not $T, \mathcal{I} \vDash \phi$,
- $T, \mathcal{I} \vDash \forall x : \phi$ iff for all $s \in S$, $T, \mathcal{I}[x := s] \vDash \phi$.

Where we define $\mathcal{I}[x := s](y) = (x = y)?s : \mathcal{I}(y)$. We also assume that constants have a fixed interpretation. If we have $T, \mathcal{I} \vDash \phi$ for all $\mathcal{I}$, we just write $T \vDash \phi$. As a sample formula, $\exists s, s' : I(s) \wedge Post^*(s, s') \wedge p(s')$ states that p holds in some state reachable from an initial state. Other relational operators can be straightforwardly included in the logic as done in [17].

Linear-Time Temporal Logic. Linear-time temporal logic (LTL) is a well-known logical formalism that has been successfully used to specify the temporal properties of concurrent and reactive systems [5]. Given a set AP of atomic propositions, the syntax of LTL is inductively defined, using the standard logical and temporal operators, by the following grammar: $\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \bigcirc\phi \mid \phi \mathbf{U} \phi$, where $p \in AP$. Additional operators, such as *true*, *false*, $\Diamond\phi$ (*eventually* ϕ), $\Box\phi$ (*always* ϕ) and $\phi \mathbf{W} \psi$ (ϕ (weak) until ψ), can be defined as combinations of the above operators [5]. LTL formulas are interpreted over infinite sequences of Boolean assignments for variables in AP, i.e., each sequence is of the form $\pi = \sigma_0\sigma_1\sigma_2 \cdots \in (2^{AP})^\omega$. The interpretation of LTL formulas can be given as a satisfaction relation $\vDash$, whose definition is standard. LTL\X [5] is the fragment of LTL obtained by omitting the operator $\bigcirc$ from formulas, typically used for expressing LTL stutter invariant properties of concurrent systems [23]. We employ LTL\X to specify global properties of distributed systems. Given an LTS T and a LTL formula ϕ, we say that $T \vDash \phi$ iff $L(\pi) \vDash \phi$ for every $\pi \in Traces(T)$.

Process Descriptions. While transition systems provide a low-level description of reactive system behavior, such systems are typically better described as collections of interacting processes,

as in languages like Promela. For convenience, we use a simple guarded-command language, as those defined in [3, 11]. A system consists of a collection of shared variables and a collection of processes that execute concurrently asynchronously. Each process has its own collection of local variables, an initialization condition indicating in which (local) state each process starts execution, and a set of guarded actions that establish how a process can proceed execution. Each atomic action has the form $[a]B \rightarrow C$ where: a is the name of the action, B (called the guard) is a Boolean statement over the (shared and local) variables of the program, and C is a collection of assignments, written $x_0{:=}E_0, \ldots, x_n{:=}E_n$, where the x_i's are local or shared variables, and E_i's are expressions of the correct type. We assume that the test of B and the assignments in C are all executed atomically. The operational semantics of this process notation is simple. A state is an assignment of values to the variables in the program. An action $B \rightarrow C$ is enabled in a state if B is true in that state. An execution of a given process is a sequence of states satisfying the following conditions: (i) the initial state satisfies the initial conditions, (ii) if the state s_i follows from s_{i-1} in the sequence, then we have some action $B \rightarrow C$ such that B holds at s_{i-1} and the execution of C results in s_i, (iii) the sequence is infinite (we assume that when no action is enabled, then the last state is repeated infinitely times). By this semantics, mapping process executions to LTS traces is straightforward, and so is the definition of satisfaction of LTL properties by system/process executions.

3 Motivating Example

We illustrate our technique with a well-known example of a distributed system, Dijkstra's *dining philosophers* [10]. Consider n philosophers sitting around a table and sharing n forks. Each philosopher has exactly two forks available, one on each side, shared with the adjacent philosophers. On the table, there is a bowl of pasta, and the philosophers alternate between thinking and eating. Each philosopher needs two forks to eat, and forks are (of course) mutually exclusive between their two users. We aimed to design a distributed protocol for this setting that guarantees deadlock freedom. As is well known, this problem admits several solutions. For instance, a way to avoid deadlock is to follow the policy that philosophers only pick up forks when both of their adjacent forks are free. Another possible solution can be obtained if the first $n-1$ philosophers follow a specific order to pick up the forks (for example, each philosopher first takes her left fork and then her right one), while the remaining philosopher takes the forks in the opposite order. These are examples of distributed protocols, that is, the philosophers operate in a decentralized manner without the help of a central entity (like a waitress). These are the kinds of solutions we expect our technique to synthesize. Let us describe how this problem is specified in our setting. The behavior of philosopher i (for $0 \leq i < n$) is modeled by the following set PS^i of formulas:

(1) $\forall s : I(s) \Rightarrow thk^i(s) \wedge \neg ownFork_i^i(s) \wedge \neg ownFork_{i+1}^i(s)$,
(2) $\forall s, s' : \neg eat^i(s) \Rightarrow \neg getThk^i(s, s')$,
(3) $\forall s, s' : getThk^i(s, s') \Rightarrow thk^i(s') \wedge \neg ownFork_i^i(s') \wedge \neg ownFork_{i+1}^i(s')$,
(4) $\forall s, s' : \neg thk^i(s) \Rightarrow \neg getHgr^i(s, s')$,
(5) $\forall s, s' : getHgr^i(s, s') \Rightarrow hgr^i(s')$,

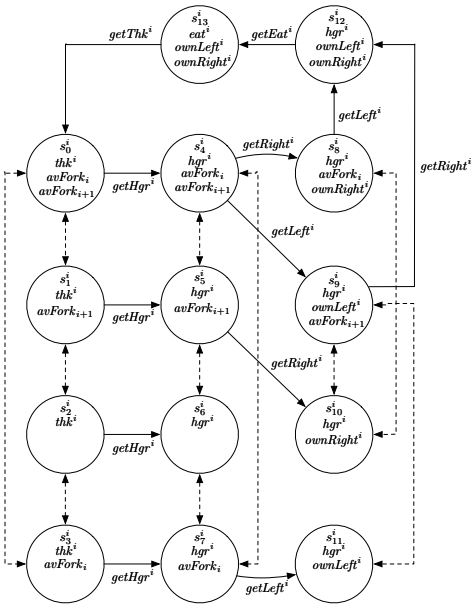


Figure 1: LTS T^i modeling philosopher i.

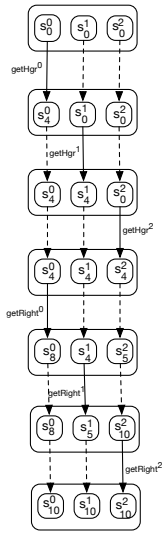


Figure 2: A Path in $T^0 \parallel T^1 \parallel T^2$ leading to deadlock.

(6) $\forall s, s' : \neg(hgr^i(s) \wedge ownFork_i^i(s) \wedge ownFork_{i+1}^i(s)) \Rightarrow \neg getEat^i(s, s')$,
(7) $\forall s, s' : getEat^i(s, s') \Rightarrow eat^i(s')$,
(8) $\forall s, s' : \neg(avFork_{i+1}(s) \wedge hgr^i(s)) \Rightarrow \neg getRight^i(s, s')$
(9) $\forall s, s' : getRight^i(s, s') \Rightarrow ownFork_{i+1}^i(s')$,
(10) $\forall s, s' : \neg(avFork_i(s) \wedge hgr^i(s)) \Rightarrow \neg getLeft^i(s, s')$
(11) $\forall s, s' : getLeft^i(s, s') \Rightarrow ownFork_i^i(s')$,
(12) $\forall s : (eat^i(s) \wedge \neg thk^i(s) \wedge \neg hgr^i(s)) \vee (\neg eat^i(s) \wedge thk^i(s) \wedge \neg hgr^i(s)) \vee (\neg eat^i(s) \wedge \neg thk^i(s) \wedge hgr^i(s))$,
(13) $\forall s \in S : I(s) \Rightarrow (\exists s' \in S : Post^*(s, s') \wedge eat^i(s'))$.

Notice how this specification captures a local view of the system. Each philosopher uses Boolean variables $ownFork_i^i$ and $ownFork_{i+1}^i$ to signal acquisition of the right and left fork, respectively. Variables $avFork_i$ and $avFork_{i+1}$ are used to indicate whether a fork is busy or not (here + denotes the addition modulo n); these are shared variables and so no superindex is needed to distinguish them. The actions of philosopher i are: $getThk^i$ (the philosopher goes to the thinking state), $getHgr^i$ (the philosopher gets hungry), $getEat^i$ (the philosopher goes to the eating state). Philosophers also have actions to gain access to their forks: $getRight^i$ (obtain the right fork if available) and $getLeft^i$ (obtain the left fork if available).

Formulas (1)-(11) model the behavior of actions in a pre/post-condition style. Formula (2) states an enabling condition for action *getThk*: it can only be executed when the current philosopher is in the eating state. On the other hand, Formula (3) expresses the postcondition of action *getThk*. Formulas (3)-(11) express similar conditions for the other actions. Formula (12) states an invariant of the system: a philosopher can only be either eating, thinking, or hungry, and these states are mutually exclusive. Finally, formula (13) is a (local) reachability constraint, stating that the philosopher i can reach the eating state from the initial state.

It is worth noting that forks act as locks: two philosophers cannot hold onto the same fork. Furthermore, forks are shared variables that can thus be changed by the "environment" (i.e., other philosophers in this case). The behavior of the environment can be modeled using additional actions: $chFork_i$ and $chFork_{i+1}$ that model the acquisition of locks by other philosophers. We may consider additional axioms to specify the behavior of these actions:

(14) $\forall s : ownFork_i^i(s) \Rightarrow \neg avFork_i(s)$,
(15) $\forall s : ownFork_{i+1}^i(s) \Rightarrow \neg avFork_{i+1}(s)$,
(16) $\forall s : \neg ownFork_i^i(s) \equiv (\exists s' : chFork_i(s, s'))$,
(17) $\forall s : \neg ownFork_{i+1}^i(s) \equiv (\exists s' : chFork_{i+1}(s, s'))$,
(18) $\forall s, s' : chFork_i(s, s') \Rightarrow (avFork_i(s) \equiv \neg avFork_i(s'))$,
(19) $\forall s, s' : chFork_{i+1}(s, s') \Rightarrow (avFork_{i+1}(s) \equiv \neg avFork_{i+1}(s'))$.

Constraints (14)-(15) state that if a philosopher owns a fork, then it cannot be grabbed by another philosopher. In contrast, formulas (16)-(17) express that if a philosopher does not own a fork, then another philosopher can grab it. Finally, formulas (18)-(19) state that actions $chFork_i$, $chFork_{i+1}$ affect the availability of the forks.

Typically, these kinds of specifications also include *frame axioms* [21], which list the variables that remain unchanged after the execution of a given action. For example, formula:

(20) $\forall s, s' : chFork_i(s, s') \Rightarrow avFork_{i+1}(s) \equiv avFork_{i+1}(s')$,

expresses that the value of variable $avFork_{i+1}$ remains unchanged after the execution of $chFork_i$.

These formulas may originate from a higher-level specification notation. We do not address the design of such a specification language here and instead refer directly to the set of formulas capturing the components' behaviors.

The inputs to our technique include a *global* temporal requirement that constrains the executions of the resulting system. For instance, in the case $n = 3$ we will have the local specifications PS^0, PS^1, and PS^2 together with the LTL formula:

(21) $\Box\neg(ownFork_1^0 \wedge ownFork_2^1 \wedge ownFork_0^2) \wedge \neg(ownFork_0^0 \wedge ownFork_1^1 \wedge ownFork_2^2)$,

which states that the system is deadlock-free. This formula is evaluated over traces of states, that is, given a trace of states $\sigma_0, \sigma_1, \ldots$, proposition p is true at instant i if $p(\sigma_i)$ holds.

Note that for each specification PS^i, a model finder can be used to build an LTS T^i that satisfies it. Given the expressiveness of the logic, this can only be done up to a given bound on the size of the LTSs (the logic is undecidable). We use the Alloy Analyzer [17], a bounded model finder for relational logic (which subsumes first-order logic with transitive closure), for this task. Fig. 1 shows an LTS satisfying PS^i, obtained in this way. The dotted arrows represent environment transitions (which, once components are composed, will become actions performed by other philosophers). The global system behavior is obtained by the (asynchronous) parallel composition of the obtained LTSs, denoted $T^0 \parallel T^1 \parallel T^2$. Now, we can verify whether the global system satisfies the global properties. In our example, if the LTSs T^0, T^1, and T^2 are as shown in Fig. 1, then the global system will contain an execution leading to deadlock. The trace in Fig. 2, in which the three philosophers get hungry in turns and then take their corresponding right fork, leads to a deadlock.

Our approach exploits counterexamples of global properties to guide the search for local implementations. For example, we force at least one of the local implementations to avoid the consecutive execution of $getHgr^i$ and $getRight^i$. Assuming that, under this new constraint, we obtain an LTS T'^0 for philosopher 0 that takes the left fork before taking the right one. The global system $T'^0 \parallel T^1 \parallel T^2$ is deadlock-free and satisfies the global temporal requirement.

4 System Specifications

In this section, we provide a more formal and detailed definition of specifications, as introduced in the previous section. In the following definitions, we mainly use first-order logic with transitive closure as described in Section 2. We begin by providing a precise definition of the concept of component specification.

Definition 4.1. *A process (or component) specification PS is a tuple $\langle\langle Sh, Loc, Act, Cons\rangle, \Phi\rangle$ where Sh, Loc, Act, Cons are finite and mutually disjoint sets, and Φ is a finite set of first-order (relational) formulas over the vocabulary defined by unary predicates $Sh \cup Loc$, binary predicates Act, and constants Cons.*

Intuitively, *Sh* refers to the shared variables used by the process, *Loc* contains the process's local variables, and *Act* names the process's actions. A process specification defines a collection of LTSs that satisfy the requirements of the specification. Given a process specification $PS = \langle\langle Sh, Loc, Act, Cons\rangle, \Phi\rangle$ and an LTS $T = \langle S, Act, \rightarrow, I, Sh \cup Loc, L\rangle$, we write $T \vDash PS$ iff $T \vDash \phi$ for every $\phi \in \Phi$.

Example 4.1. *For the example of Section 3, we have that $Sh = \{avFork_i, avFork_{i+1}\}$, $Loc = \{eat^i, thk^i, hgr^i, ownFork_i^i, ownFork_{i+1}^i\}$, $Act = \{getThk^i, getHgr^i, getEat^i, getLeft^i, getRight^i, chFork_i, chFork_{i+1}\}$, and $Cons = \emptyset$. The formula is the conjunction of formulas (1)-(19) (plus the frame axioms not shown there).*

A specification of a distributed system is a collection of process specifications over the same shared variables, and an additional global temporal requirement.

Definition 4.2. *A system* specification $\mathcal{S}$ *is a tuple* $\langle \{PS^i\}_{i\in\mathcal{I}}, \phi\rangle$*, where* $\mathcal{I}$ *is a finite index set, each* $PS^i = \langle\langle Sh^i, Loc^i, Act^i, Cons\rangle, \Phi^i\rangle$ *is a* local *process specification, and* ϕ *is an LTL formula over the vocabulary* $\bigcup_{i\in\mathcal{I}} Sh^i \cup \coprod_{i\in\mathcal{I}} Loc^i$ *expressing a global requirement.*

Example 4.2. *Taking again the example of the philosophers, we can consider three instances of the specification for* $i \in \{0,1,2\}$ *whose sets* Sh^i*,* Loc^i*,* Act^i*, and* $Cons^i$ *are as in Example 4.1, and the global requirement is formula (21).*

By simply treating a system specification $\mathcal{S}$ as a collection of component specifications, we do not provide specific semantics for shared variables compared to process-local variables. As a result, locally correct implementations may often lead to invalid system implementations — those that violate the global constraints — because local assumptions about shared variables may not hold when the processes are combined. This can be solved by forcing local process specifications to assume unrestricted behavior of the environment, as formalised in the following definition.

Definition 4.3. *Let* $\mathcal{L} = \{\ell_0, \ldots, \ell_m\}$ *be a set of elements (called* locks*), and* $Av_{\mathcal{L}} = \{av_\ell \mid \ell \in \mathcal{L}\}$ *be a set of variables.*

An $\mathcal{L}$-synchronized specification *is a system specification* $\mathcal{S} = \langle \{PS^i\}_{i\in\mathcal{I}}, \phi\rangle$ *with* shared *variables* $Av_{\mathcal{L}} \subseteq \bigcup_{i\in\mathcal{I}} Sh^i$*, such that for each process specification* $PS^i = \langle\langle Sh^i, Loc^i, Act^i, Cons^i\rangle, \Phi^i\rangle$*, it holds that:*

(1) if $av_\ell \in Sh^i \cap Av_{\mathcal{L}}$*, then* $own_\ell \in Loc^i$ *and* $ch_\ell \in Act^i$*,*
(2) if $g \in Sh^i \setminus Av_{\mathcal{L}}$*, then* $ch_g \in Act^i$*.*

Furthermore, the following formulas belong to each Φ^i*:*

(a) $\bigwedge_{av_\ell \in Sh^i \cap Av_{\mathcal{L}}} (\forall s : own_\ell(s) \Rightarrow \neg av_\ell(s))$,
(b) $\bigwedge_{av_\ell \in Sh^i \cap Av_{\mathcal{L}}} (\forall s : \neg own_\ell(s) \equiv (\exists s' : ch_\ell(s,s')))$,
(c) $\bigwedge_{av_\ell \in Sh^i \cap Av_{\mathcal{L}}} (\forall s,s' : ch_\ell(s,s') \Rightarrow (av_\ell(s) \equiv \neg av_\ell(s')))$,
(d) $\bigwedge_{av_\ell \in Sh^i \cap Av_{\mathcal{L}}} \bigwedge_{v \in (Sh^i \cup Loc^i \setminus \{own_\ell, av_\ell\})} (\forall s,s' : ch_\ell(s,s') \Rightarrow (v(s) \equiv v(s')))$,
(e) $\bigwedge_{g \in (Sh^i \setminus Av_{\mathcal{L}})} (\forall s : (\exists s' : ch_g(s,s') \wedge g(s')) \wedge (\exists s' : ch_g(s,s') \wedge \neg g(s')))$,
(f) $\bigwedge_{g \in (Sh^i \setminus Av_{\mathcal{L}})} \bigwedge_{g' \in (Sh^i \cup Loc^i) \setminus \{g\}} (\forall s,s' : ch_g(s,s') \Rightarrow (g'(s) \equiv g'(s')))$.

Some words about the above definition are useful. Locks are a special type of shared variable used for synchronization. Actions $\{ch_{\ell_0}, \ldots, ch_{\ell_m}\} \cup \{ch_g \mid g \in Sh^i \setminus Av_{\mathcal{L}}\}$ are used to model environment actions; for instance, ch_{ℓ_0} represents an environment action that changes the state of lock ℓ_0. The variables av_ℓ and own_ℓ are used to indicate that lock ℓ is free or owned by the current component, respectively. Formula (a) indicates that if a component owns a lock, then the lock is not available for any other component. Formula (b) expresses that if a lock is not owned by the current component, then the lock can be changed by the environment. Formula (c) says that if the environment changes lock ℓ_i, then the variable av_{ℓ_i} changes accordingly. Formula (d) states that changes in locks do not affect other variables. Formula (e) expresses that shared variables can be changed by the environment. Finally, formula (f) states that the changes in one shared variable do not affect the other variables. From now on, we write $s \dashrightarrow s'$ if $s \xrightarrow{ch_\ell} s'$ or $s \xrightarrow{ch_g} s'$, for some lock ℓ or shared variable g, respectively.

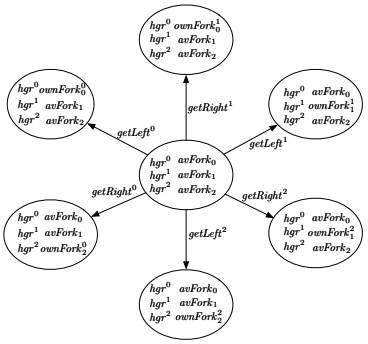


Figure 3: Fragment of the Asynchronous product $T^0 \parallel T^1 \parallel T^2$.

Example 4.3. *For the system specification of Example 3 with* $i \in \{0,1,2\}$*, we have that* $\mathcal{L} = \{Fork_0, Fork_1, Fork_2\}$*, and the corresponding variables* $Av_{\mathcal{L}} = \{avFork_0, avFork_1, avFork_2\}$*. Furthermore,* $\{avFork_i, avFork_{i+1}\} \subseteq Sh^i$ *and actions* $\{chFork_i, chFork_{i+1}\} \subseteq Act^i$ *(for any i). In this example, formulas (14)-(15) correspond to formula (a), formulas (16)-(17) correspond to formula (b), and formulas (18)-(19) correspond to formula (c). Formula (d) corresponds to frame axioms (e.g., formula (20)). Formulas (e) and (f) are not instantiated in this example (the forks are the unique shared variables).*

In the same way that component specifications can be put together, we also consider the asynchronous composition of LTSs.

Definition 4.4. *Let* $\mathcal{S} = \langle \{PS^i\}_{i\in[0,n]}, \phi\rangle$ *be an* $\mathcal{L}$-synchronized specification*, and let* $T^0, \ldots, T^n$ *be LTSs, such that for each* $T^i = \langle S^i, Act^i, \rightarrow^i, I^i, Sh^i \cup Loc^i, L^i\rangle$ *we have* $T^i \vDash PS^i$*. We define the LTS* $T^0 \parallel \cdots \parallel T^n = \langle S, \coprod_{i\in[0,n]} Act^i, \rightarrow, I, Sh \cup \coprod_{i\in[0,n]} Loc^i, L\rangle$ *as follows:*

- $S = \{s \mid s \in \prod_{i\in[0,n]} S^i \wedge (\forall i,j \in [0,n] : g \in Sh^i \cap Sh^j \Rightarrow L^i(s{\restriction}i)(g) = L^j(s{\restriction}j)(g))\}$,
- $\rightarrow = \{s \xrightarrow{a} s' \mid \exists i \in [0,n] : (s{\restriction}i \xrightarrow{a}{}^{i} s'{\restriction}i) \wedge (\forall j \neq i : (s{\restriction}j \dashrightarrow^j s'{\restriction}j) \vee (s{\restriction}j = s'{\restriction}j))\}$,
- $I = \{s \mid \forall i \in [0,n] : s{\restriction}i \in I^i\}$,
- $L(s)(x) = \begin{cases} L^i(s{\restriction}i)(x) & \text{if } x \in Loc^i \text{ for some } i \in [0,n], \\ L^0(s{\restriction}0)(x) & \text{otherwise.} \end{cases}$

Example 4.4. *For the dining philosophers example, we can consider the asynchronous product of the LTSs* T^0, T^1, T^2 *(as described in Fig. 1). The resulting structure has more than 60 states. Fig. 3 shows a small fragment of the resulting LTS: the state where all the philosophers are hungry and the outgoing transitions from this state.*

Now, we can define the semantics of distributed specifications using asynchronous products, i.e., the combination of LTSs that produces all interleavings of their corresponding actions.

Definition 4.5. *Given a system specification* $\mathcal{S} = \langle \{PS^i\}_{i\in[0,n]}, \phi\rangle$ *a* model *or* implementation *of* $\mathcal{S}$ *is a collection of LTSs* $T^0, \ldots, T^n$ *such that* $T^i \vDash PS^i$*, for every* $i \in [0,n]$*, and* $T^0 \parallel \cdots \parallel T^n \vDash \phi$*.*

Given an LTS T, we can obtain a specification that characterizes it (up to isomorphism), using fresh constants to identify the states, and formulas to describe the transitions. Moreover, we can obtain

a specification that characterizes *all* LTSs that can be obtained by removing some (local) transitions from T. Intuitively, this specification captures the possible refinements of T, in the sense that some non-determinism is removed.

Definition 4.6. *Let $PS = \langle\langle Sh, Loc, Act, Cons\rangle, \Phi\rangle$ be a component specification, and let $T = \langle S, Act, \rightarrow, I, (Sh \cup Loc), L\rangle$ be an LTS, such that $S = \{s_0, \ldots, s_n\}$ (without loss of generality we assume $S \cap Cons = \emptyset$), $I = \{s_0, \ldots, s_k\} \subseteq S$,$Sh \cup Loc = \{p_0, \ldots, p_m\}$, $Act = \{a_0, \ldots, a_k\}$ and $T \vDash PS$. The process specification $Ref(PS, T)$ is the tuple $\langle\langle Sh, Loc, Act, Cons \cup S\rangle, \Phi \cup \{\phi^T\}\rangle$ where ϕ^T is the following formula:*

$$
\begin{aligned}
&\Big(\bigwedge_{0\le i<j\le n} s_i \neq s_j\Big) \wedge \bigwedge_{i\in[0,q]} I(s_i) \\
&\qquad \wedge \bigwedge_{j\in[0,n]\wedge i\in[0,m]\wedge p_i\in L(s_j)} p_i(s_j) \\
&\qquad \wedge \bigwedge_{j\in[0,n]\wedge i\in[0,m]\wedge p_i\notin L(s_j)} \neg p_i(s_j) \\
&\qquad \wedge \bigwedge_{j,j'\in[0,n]\wedge i\in[0,k]\wedge\neg(s_j \xrightarrow{a_i} s_{j'})} \neg a_i(s_j, s_{j'}) \\
&\qquad \wedge \bigwedge_{j,j'\in[0,n]\wedge i\in[0,k]\wedge s_j \xrightarrow{a_i} s_{j'}\wedge a_i\in\{ch_g \mid g\in Sh\}} a_i(s_j, s_{j'})
\end{aligned}
$$

Roughly speaking, the formula ϕ^T identifies each state with a constant, describes the properties of each state, enumerates the transitions labeled with environment actions, and rules out the introduction of local transitions not present in T. Furthermore, note that the models of $Ref(PS, T)$ may remove some transitions of T but still satisfy the specification PS.

The following property will be important for Section 6.

Property 4.1. *Let $PS = \langle\langle Sh, Loc, Act, Cons\rangle, \Phi\rangle$ be a process specification, and T an LTS such that $T \vDash PS$ then:*

- *For any LTS T', if $T' \vDash Ref(PS, T)$, then $T' \vDash PS$,*
- *$T \vDash Ref(PS, T)$.*

The following notation will be helpful in later sections to refer to specifications augmented with additional formulas.

Definition 4.7. *Let $PS = \langle\langle Sh, Loc, Act, Cons\rangle, \Phi\rangle$ be a process specification, and $T = \langle S, Act, \rightarrow, I, (Sh \cup Loc), L\rangle$ an LTS such that $S = \{s_0, \ldots, s_n\}$, and $T \vDash PS$. Given a formula ψ over the language of T, we define:*

$$Ref(PS, T) \oplus \psi = \langle\langle Sh, Loc, Act\rangle, \Phi \cup \{\phi^T, \psi\}\rangle$$

Counterexamples. Let us introduce some definitions and concepts related to counterexamples that will be useful in the subsequent sections.

In finite transition structures, counterexamples can be represented by finite paths, called *lasso traces* (finite sequences of states such that the last state loops back to a previous state) [6]. Noting that any finite path can be identified with a formula in *conjunctive normal form* yields the following definition.

Definition 4.8. *Given an LTS $T = \langle S, Act, \rightarrow, I, AP, L\rangle$ and a finite path $\pi = s_0, s_1, \ldots, s_m \in Path(T)$, we define the formula: $\mathsf{CNF}(\pi) = \bigwedge_{0\le j<m}(\bigvee_{a\in Act}(a(s_j, s_{j+1}))$, where $s_0, \ldots, s_m$ are constants.*

Note that $Ref(PS, T) \oplus \mathsf{CNF}(\pi)$ captures the refinements of T satisfying specification PS and preserving the path π. Similarly, we can define a formula that removes a counterexample from the instances of a specification.

Definition 4.9. *Let $T = \langle S, Act, \rightarrow, I, AP, L\rangle$ be an LTS and $\pi = s_0, s_1, \ldots, s_m \in Path(T)$ a path such that $s_i \neq s_{i+1}$ for some i. We define the following formula over T: $\mathsf{NOT}(\pi) = \bigvee_{0\le i<m}(\bigwedge_{a\in Act} \neg a(s_i, s_{i+1}) \wedge (s_i \neq s_{i+1}))$, where $s_0, \ldots, s_m$ are constants.*

$Ref(PS, T) \oplus \mathsf{NOT}(\pi)$ identifies the refinements of T that do not contain path π. The proof of the following theorem is direct from the above definitions.

Theorem 4.1. *Let PS and T be a specification and an LTS, respectively, such that $T \vDash PS$, and $\pi = s_0, \ldots, s_m \in Path(T)$, with $s_i \neq s_{i+1}$ for some i. Then, $Ref(PS, T) \oplus \mathsf{NOT}(\pi) \nvDash Ref(PS, T) \oplus \mathsf{CNF}(\pi)$.*

Furthermore, given LTSs $\{T^i\}_{i\in I}$ and a finite path $\pi = s_0, \ldots, s_m \in Path(\|_{i\in I}\ T^i)$, we denote by $\pi\!\uparrow\! i$ the path projected to an execution of the process T^i, defined as $\pi\!\uparrow\! i = s_0\!\uparrow\! i, \ldots, s_m\!\uparrow\! i$. Note that projecting a global execution may introduce stuttering steps in the obtained local executions. The projections of a global execution form a tuple $\langle\pi\!\uparrow\! i\rangle_{i\in I}$, which will be used to refine the instances obtained via the model finder.

5 Obtaining Guarded-Command Programs.

This section describes the methodology we use to obtain guarded command programs from LTSs. It is worth noting that the programs that we synthesize use locks to achieve synchronization. We assume a non-blocking semantics for locks; that is, if a guard checks for the availability of a lock and the lock is unavailable, the process may proceed with the execution of alternative branches. However, note that a process gets blocked when all its guards become false. Other synchronization mechanisms, such as blocking locks, semaphores, and condition variables, can be expressed by our guarded command notation.

Given LTSs $T^i = \langle S^i, Act^i, \rightarrow^i, I^i, Sh^i \cup Loc^i, L^i\rangle$ for $i \in [0, n]$, we can abstract away the environmental transitions and define a corresponding program in guarded command notation, denoted $\mathrm{Prog}(T^0 \parallel \cdots \parallel T^n)$, as follows. The shared variables are those in Sh^i plus an additional shared variable ℓ for each lock, whose domain is $[0, n] \cup \{\bot\}$ ($\bot$ is a value indicating that the lock is free). Additionally, for each T^i we define a corresponding process. To do so, we introduce for each $s \in S^i$ the equivalence class: $[s] = \{s' \in S^i \mid (s \dashrightarrow^* s') \vee (s' \dashrightarrow^* s)\}$. That is, it is the set of states connected to s via environmental transitions, which is already an equivalence class. The collection of all equivalence classes is denoted $S^i/_{\dashrightarrow^*}$. The local variables of the process are those in Loc^i plus a fresh variable st_i, with domain $S^i/_{\dashrightarrow^*}$ (to indicate the current state of the process).

Finally, given states $s, s' \in S^i$ with $s \xrightarrow{a} s' \in \rightarrow^i$ and $[s] \neq [s']$, we consider the guarded command shown in Fig. 4. We can prove that our translation from transition systems to programs is correct, i.e., both satisfy the same temporal properties. This can be proven by verifying that both generate the same traces modulo a translation of temporal properties, defined as follows: $\theta(own_\ell) = (\ell = i)$, $\theta(av_\ell) = (\ell = \bot)$, $\theta(p) = p$ (for any $p \in Loc \cup Sh$),

$$[a]\left(\begin{array}{l} state = [s] \\ \wedge \bigwedge\{x = x(s) \mid x \in Sh^i \cup Loc^i\} \\ \wedge \bigwedge\{\ell = i \mid \ell \in \mathcal{L} : own_\ell \in L^i(s)\} \\ \wedge \bigwedge\{\ell = \bot \mid \ell \in \mathcal{L} : av_\ell \in L^i(s)\} \end{array}\right) \rightarrow \left(\begin{array}{l} \{x{:=}x(s') \mid x \in Loc \cup Sh\} \\ \cup\{state{:=}[s']\} \\ \cup\{\ell := i \mid \ell \in \mathcal{L} : own_\ell \in L^i(s)\} \\ \cup\{\ell := \bot \mid \ell \in \mathcal{L} : av_\ell \in L^i(s)\} \end{array}\right)$$

Figure 4: Guarded command constructed from an LTS transition

Program *Mutex*
  var m : *Lock*;
  Process P^i *with* $i \in \{0, 1\}$
    var try^i, ncs^i, cs^i : *bit*
    var st^i : $\{S0, S1, S2, S3, S4, S5\}$
    initial : $ncs^i \wedge \neg cs^i \wedge \neg try^i$
    begin
      $[enterTry]$ $st^i = S5 \rightarrow st^i := S3, try^i := 1, ncs^i := 0$
      $[getLock]$ $st^i = S3 \wedge m = \bot \rightarrow st^i := S1, try^i := 1, m := i$
      $[enterCS]$ $st^i = S1 \rightarrow st^i := S0, cs^i := 1, try^i := 0$
      $[enterNCS]$ $st^i = S0 \rightarrow st^i := S5, ncs^i := 1, cs^i := 0, m := \bot$
    end
end

Figure 6: Program for Mutex

$\theta(\neg\varphi) = \neg\theta(\varphi)$, $\theta(\varphi \vee \psi) = \theta(\varphi) \vee \theta(\psi)$, $\theta(\bigcirc\psi) = \bigcirc\theta(\psi))$, and $\theta(\phi \mathbf{U} \psi) = \theta(\phi)\mathbf{U}\theta(\psi)$.

Theorem 5.1. *Given LTSs T^i for $i \in [0, n]$ and LTL property ϕ, we have that: $T^0 \parallel \cdots \parallel T^n \vDash \phi \Leftrightarrow Prog(T^0 \parallel \cdots \parallel T^n) \vDash \phi$*

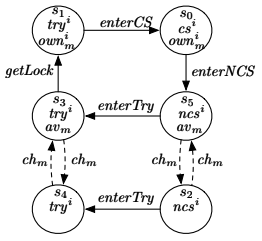


Figure 5: LTS for Mutex

Example 5.1 (Mutex). *Consider a system composed of two processes (which can be straightforwardly generalized to n processes), both with non-critical, waiting, and critical sections. The global property is mutual exclusion: the two processes cannot be in their critical sections simultaneously. We consider one lock m, actions: enterNCS^i (the process i enters the non-critical section), enterCS^i (the process i enters the critical section), getLock^i (the process acquires the lock), enterTry^i (the process goes to the try state), and the corresponding propositions $ncs^i, cs^i, try^i, own^i_m, av_m$.*

The transition system is shown in Fig. 5 and the corresponding program is depicted in Fig. 6. It is interesting to observe that the conditions in Def. 4.3 allow the use of locks as synchronization mechanisms. If component i owns the lock m, then the other processes cannot enter their critical sections because the lock is unavailable to them. Also, note that we consider one state in the program for each equivalence class of states in the LTS.

6 Synthesis Algorithm

In this section, we describe our synthesis algorithm. Given a specification $\mathcal{S} = \langle\{PS^i\}_{i\in\mathcal{I}}, \phi\rangle$ for a distributed system, the algorithm synthesizes process implementations $\{T^i\}_{i\in\mathcal{I}}$ that satisfy their local specifications, i.e., $T^i \models PS^i$, for all $i \in \mathcal{I}$. Furthermore, the asynchronous parallel composition of these processes satisfies the global temporal requirement ϕ, i.e., $\|_{i\in\mathcal{I}}\ T^i \models \phi$. The algorithm uses a bound k to limit the maximum number of states for the synthesized LTSs. If a valid implementation is found, it is returned in NuSMV language. Otherwise, the algorithm labels the specification as unsatisfiable within the bound k.

Alg. 1 presents a basic version of the synthesis procedure. By executing **Synt**$(0, PS^0, \ldots, PS^n, k)$, the program recursively inspects all the specifications and synthesizes, if possible, a solution. In line 2, Alg. 1 inspects all instances of the current specification. Line 4 handles the base case (i.e., the last specification), where a model checker is called to verify the selected instances. If successful, a solution is returned; otherwise, another instance is chosen. Line 6 addresses the recursive case, in which a different instance is examined, and the algorithm continues recursively with the next specification. As shown in Section 7, this algorithm can only handle problems of small size.

Algorithm 1: A Simple Search Algorithm

```
   Data: i (process number) PS^0, ..., PS^n (process specifications), φ
         (global property), k (instance bound)
1  Function Synt(i, PS^0, ..., PS^n, k)
       Result: r_0, ..., r_n with r_i ⊨ PS^i and r_0 ∥ ··· ∥ r_n ⊨ φ or ∅
               otherwise
2      for all instances M_i of PS^i do
3          r_i ← M_i;
4          if i = n ∧ r_0 ∥ ··· ∥ r_n ⊨ φ then return {r_0, ..., r_n};
5          if i < n then
6              found ← Synt(i + 1, PS^0, ..., PS^n, k);
7              if found ≠ ∅ then return found;
8      return ∅;
```

A key issue with Alg. 1 is that an incorrect choice for the initial instance of the first specifications (e.g., PS^0) may cause the algorithm to stall on later specifications (e.g., PS^n), before returning to the initial specifications and trying with different instances for them.

We improve Alg. 1 using mainly three techniques. First, we force the synthesizer to start from "better" initial instances (see below). Second, we use counterexamples to accelerate the search. Third, we use batches to prevent the procedure from stalling on the last components.

For the first point, we augment the specifications with formulas that ensure the synthesized instances are not overly restrictive with respect to transitions, which can be pruned in later stages.

Algorithm 2: Batches Algorithm

```
Data: PS^0, ..., PS^n (process specifications), φ (global property), k
      (instance size bound), b_0, ..., b_m (sequence of batch bounds),
      cexs (global set of counterexamples)
1  Function StartSearch(PS^0, ..., PS^n, k, b_0, ..., b_m)
       Result: r_0, ..., r_n with r_i ⊨ PS^i and r_0 ∥ ··· ∥ r_n ⊨ φ or ∅
               otherwise
2      cexs ← ∅;
3      foreach i = 0 ... n do M_i ← initial instance of PS^i of size k;
4      for b = b_0, ..., b_m do
5          foreach i = 0 ... n do
               PS^i ← Ref(PS^i, M_i) ⊕ ⋀_{π∈cexs} NOT(π↑i), ;
6          found ← BatchSynt(0, PS^0, ..., PS^n, φ,b);
7          if found ≠ ∅ then return found;
8      return ∅;
9  Function BatchSynt(i, PS^0, ..., PS^n, φ, b)
       Result: r_0, ..., r_n with r_i ⊨ PS^i and r_0 ∥ ··· ∥ r_n ⊨ φ or ∅
               otherwise
10     j ← 0;
11     while PS^i has unprocessed instances and j < b do
12         r_i ← next instance of PS^i;
13         if i = n then
14             if r_0 ∥ ··· ∥ r_n ⊨ φ then return {r_0, ..., r_n};
15             cexs ← process counterexamples;
16         if i < n then
17             found ← BatchSynt(i + 1, PS^0, ..., PS^n, φ, b);
18             if found ≠ ∅ then return found;
19         j ← j + 1 ;
20     return ∅;
```

Specifically, for every action *act*, described using pre and post-conditions (as in Section 3), we add the formula:

$$\forall s \in S : Pre_{act}(s) \Rightarrow \exists s' \in S : act(s, s'). \quad (1)$$

This formula states that, for every state in which the precondition of the action (Pre_{act}) holds, there exists at least one transition corresponding to the execution of *act*.

The first technique is instrumental to our second technique, i.e., the use of counterexamples to prune action executions leading to violations of global properties. Furthermore, to avoid the algorithm from getting stuck while inspecting instances of the latter specifications without returning to try new instances of the first specifications, we introduce the use of batches. A sequence of bounds $b_0 \dots b_n$ is used to perform a bounded backtracking. In addition, the counterexamples collected in each batch are utilized to speed up the process. Roughly speaking, we first execute the algorithm, inspecting only b_0 instances of each specification. If no solution is found, we use the collected counterexamples, now with b_1 batches. This process is repeated following the given sequence of bounds.

Alg. 2 implements the ideas discussed above. This algorithm consists of a starting function (**StartSearch**) that sets the initial models for each specification (line 3). As explained above, these initial models are obtained by augmenting the specifications with formulas instancing schema 1 and feeding them to the model finder. Line 5 initializes each specification with a tailored specification $Ref(PS^i, M_i) \oplus \bigwedge_{\pi \in cexs} \mathsf{NOT}(\pi{\uparrow}i)$. Intuitively, these specifications describe refinements of the initial LTSs that exclude the counterexamples listed in *cexs*. Line 6 calls the auxiliary function **BatchSynt**, which has a behavior similar to that of Alg. 1, but taking into account the bounds for each batch (line 4).

Soundness. The soundness of the approach follows from Property 4.1, which guarantees that local properties are preserved; and from the soundness of the model checker used, which ensures that the global property is satisfied.

Completeness. The approach is not necessarily complete, as the bounds may prevent it from inspecting some instances. However, in Section 7, we show that this procedure can handle complex examples, offering a better performance than the exhaustive approach.

7 Experimental Evaluation

We evaluate our approach around the following research questions:

RQ1 *How well does our synthesis approach deal with examples of different complexities?*

RQ2 *How good is the counterexample-guided search for speeding up the synthesis method?*

RQ3 *How do the selected bounds affect the synthesis method?*

RQ4 *How does our approach compare with related approaches?*

Experimental Setup. To answer these questions, we implemented Alg. 1 and Alg. 2 in a Java prototype tool[1]. The tool utilizes the Alloy Analyzer [17] (with Minisat) to generate instances of specifications and NuSMV [9] to verify the candidates. The experiments were conducted on an Apple M2 processor with 16GB of memory.

Benchmarks. We evaluated our approach on two groups of examples. The first group consists of eight examples of distributed algorithms: the dining philosophers (phil) [10] (our running example), Mutex (mut) [16], Readers and Writers (rw) [16], the generalized version of Peterson's algorithm (pet) [16], and the combined-tree Barrier protocol (bar) [16].

The second group consists of three variations of the ARM AMBA Advanced High Performance Bus Arbiter [2] as presented in [19, 24]: a simple arbiter (arb), a full arbiter (farb), and the Pnueli arbiter (parb). The arbiter examples assume a distributed token ring architecture, which we modeled using the Alloy language. Arbiter examples are commonly used as benchmarks for synthesis tools.

The tool's source code and data used for the experimental evaluation in the current paper are available at [8].

Tables 1 summarize the experimental results. For each example, we report the bound on the size of the processes (Sc.); the maximum time needed to generate local process instances (L.Time); the total time required to synthesize the system (G.Time); number of model checker invocations (It.) by the synthesizer; and the final result of our synthesis algorithm: 'F' (an implementation was found), 'N' (no implementation was found), 'TO' (timed out), or 'U' (the specification was found unsatisfiable by the Alloy tool). We also report the number of reachable states (R.St.), and the total states (T.St.) of the obtained implementations, expressed as powers of 2. Due to space constraints, we include only a few configurations that the solver

[1]The source code is available at https://github.com/cl-unrc-lab/PhilStone.

Table 1: Experimental Results.

Ex.	Sc.	L.Time	G.Time	It.	R.St.	T.St.	Rs.
mut(2)	4	0.22	0.46	9	$2^{2.58}$	$2^{9.33}$	F
mut(3)	4	0.20	0.87	27	2^{3}	$2^{13.50}$	F
mut(4)	4	0.20	2.94	81	$2^{3.32}$	$2^{17.67}$	F
mut(5)	4	0.20	11.05	243	$2^{3.58}$	$2^{21.84}$	F
mut(6)	4	0.20	47.63	729	$2^{3.80}$	$2^{26.01}$	F
mut(7)	4	0.21	220.86	2187	2^{4}	$2^{30.18}$	F
phil(2)	14	0.90	1.22	5	2^{4}	$2^{14.33}$	F
phil(3)	14	0.82	1.85	14	$2^{5.95}$	$2^{21.50}$	F
phil(4)	14	0.80	5.98	41	2^{8}	$2^{28.67}$	F
phil(5)	14	0.81	197.65	335	$2^{9.39}$	$2^{35.84}$	F
phil(6)	14	0.80	973.32	365	2^{12}	$2^{43.01}$	F
phil(7)	14	0.86	TO	-	-	-	TO
bar(2)	16	0.76	1.53	11	$2^{2.58}$	$2^{14.58}$	F
bar(3)	16	1.05	10.76	97	$2^{5.90}$	$2^{23.16}$	F
bar(4)	16	2.97	62.11	85	$2^{10.75}$	$2^{32.16}$	F
bar(5)	16	3.87	TO	-	-	-	TO
pet(2)	12	0.41	0.51	2	2^{2}	$2^{8.16}$	F
pet(3)	19	TO	-	-	-	-	TO
pet(3)	20	238.77	239.33	5	$2^{3.32}$	$2^{16.22}$	F
r(1)w(1)	6	0.28	0.44	7	$2^{1.58}$	2^{7}	F
r(1)w(2)	6	0.32	0.86	25	2^{2}	2^{10}	F
r(1)w(3)	6	0.36	2.11	79	$2^{2.32}$	2^{13}	F
r(1)w(4)	6	0.38	6.21	241	$2^{2.58}$	2^{16}	F
r(1)w(5)	6	0.44	19.62	723	$2^{2.80}$	2^{19}	F
r(1)w(6)	6	0.45	67.66	2181	2^{3}	2^{22}	F
r(2)w(1)	12	0.64	1.53	29	$2^{2.32}$	$2^{13.16}$	F
r(2)w(2)	12	0.89	6.98	167	$2^{2.58}$	$2^{16.16}$	F
r(2)w(3)	12	1.17	109.62	1377	$2^{2.80}$	$2^{19.16}$	F
r(2)w(4)	11	-	0.8	-	-	-	U
r(2)w(4)	12	1.56	TO	-	-	-	TO
r(3)w(1)	24	37.75	38.19	4	$2^{3.16}$	2^{13}	F
r(3)w(2)	24	74.95	76.57	13	$2^{3.32}$	2^{16}	F
r(3)w(3)	24	109.62	113.52	40	$2^{3.45}$	2^{19}	F
r(3)w(4)	24	147.42	159.07	121	$2^{3.58}$	2^{22}	F
r(3)w(5)	24	184.08	235.31	362	$2^{3.70}$	2^{25}	F
r(3)w(6)	24	220.12	382.62	1091	$2^{3.80}$	2^{28}	F
arb(2)	12	0.814	1.322	7	$2^{2.32}$	2^{12}	F
arb(3)	12	0.728	2.11	16	2^{4}	2^{18}	F
arb(4)	12	0.854	4.303	16	$2^{5.24}$	2^{24}	F
arb(5)	12	0.976	12.54	16	$2^{6.35}$	2^{30}	F
arb(6)	12	1.134	52.344	16	$2^{7.40}$	2^{36}	F
farb(2)	12	0.552	1.164	8	$2^{2.58}$	2^{12}	F
farb(3)	12	0.75	2.97	20	$2^{3.45}$	2^{18}	F
farb(4)	12	0.935	15.602	56	$2^{4.39}$	2^{24}	F
farb(5)	12	1.232	85.933	164	$2^{5.35}$	2^{30}	F
farb(6)	12	2.034	672.539	488	$2^{6.33}$	2^{36}	F
parb(2)	12	0.662	1.166	8	$2^{2.80}$	2^{12}	F
parb(3)	12	0.714	2.277	20	$2^{3.45}$	2^{18}	F
parb(4)	12	0.886	8.593	56	$2^{4.39}$	2^{24}	F
parb(5)	12	1.092	54.25	164	$2^{5.35}$	2^{30}	F
parb(6)	12	1.324	700.731	488	$2^{6.33}$	2^{36}	F

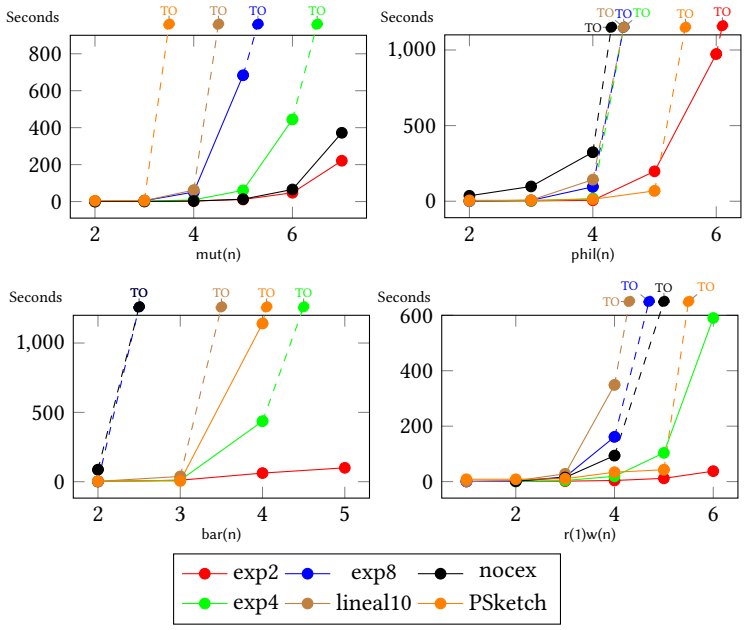


Figure 7: Comparison with PSketch

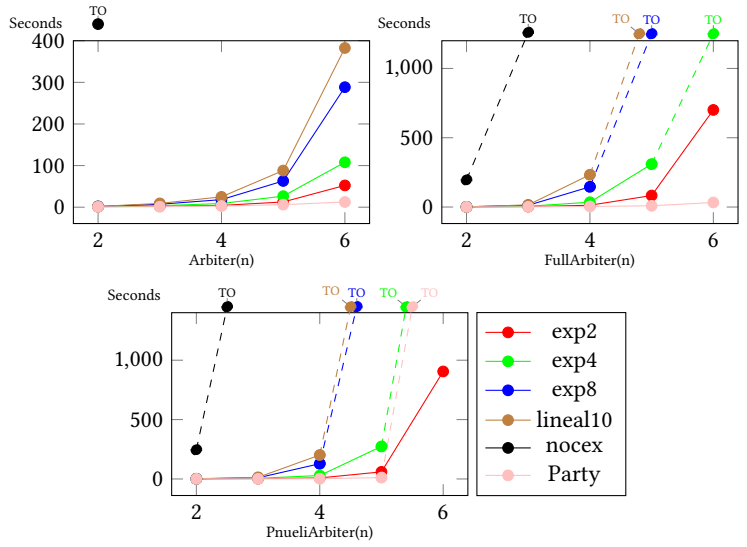


Figure 8: Comparison with Party

identified as unsatisfiable. Similar numbers can be obtained for the remaining cases if the specifications are processed with smaller scopes. We indicate the number of processes considered in each experiment, e.g., phil(6) indicates that we considered 6 concurrent processes (i.e., philosophers) in the dining philosophers example. In the case of Readers and Writers, r(n)w(m) means that n readers and m writers were considered. We set a timeout of 30 minutes.

RQ1: Effectiveness/efficiency of the approach. Table 1 shows that the technique scales reasonably well for those case studies in which each process uses a small number of locks and shared variables (e.g., dining philosophers, mutex and reader-writers). The scalability of the technique diminishes as the number of shared variables accessed by the processes increases (e.g., Peterson for n processes). Intuitively, more shared variables imply more actions performed by the environment, which increases the size of the formula fed to the SAT solver. We plan to investigate how to equip specifications with assumptions about the environment's behavior, thereby restricting the possible values of shared variables. This may simplify the SAT problem when searching for local implementations. It is worth noting that even though the algorithm is incomplete, we have not observed any "not found" outputs in our benchmarks. This could

be due to the selected timeouts. We leave an in-depth investigation of this as further work.

RQ2: Speedup using counterexamples. We have run Alg. 1 on the examples of our benchmarks. The results can be observed in Figs 7 and 8. Alg. 2 shows a much better performance in all examples. For most of them Alg. 1 ((nocex)) timed out with more of two processes. Hence, the speedup achieved using counterexamples is remarkable.

RQ3: Selection of bounds. To answer this question, we compare the results obtained with several bound configurations for the exploration phase, namely: exp4 $(4, 16, 64, \dots)$, exp8 $(8, 64, 512, \dots)$, lineal10 $(10, 20, 30, \dots)$, and for **RQ2** we also considered Alg. 1, which does not take into account counterexamples (nocex). The obtained results are depicted in Figs.7 and 8. Note that time-outs are marked using dashed lines going out of the y-axis. In general, exp2 behaves better than the other options; thus, it seems better to collect a few counterexamples first and use them to improve the search. A possible drawback of this setting is that a wrong choice of the first counterexamples may have as a consequence that no implementation is found, i.e., one may expect that this configuration is "more incomplete" than the other options. However, we have not observed this in our benchmarks.

RQ4: Comparison with related approaches. To answer this question, we included in our analysis the synthesis tool PSketch [27], which implements a Counterexample-driven Guided Inductive Synthesis (CEGIS) algorithm to obtain code from sketched code (code annotated with "holes"). To run PSketch, we took the Dining Philosophers specification provided in [27] and elaborated PSketch specifications for Mutex, Readers-Writers, and the Barrier example. The Peterson example cannot be analysed with PSketch since it only supports the analysis of safety properties. For the arbiter case studies, we compare with the tool Party [19], which is specifically tailored for distributed systems that use token ring architectures.

Our comparison focuses only on the time required by each technique for synthesizing distributed solutions. The plots of Figs. 7 and 8 depict the results of this comparison. In all the case studies, we notice that the efficiency of PSketch is drastically affected as the number of processes to synthesize increases. For instance, in the dining philosophers with 6 processes PSketch timed out. In contrast, our tool was able to obtain a solution. Similarly, in the case of 1 reader and 6 writers, PSketch failed to synthesize a solution, while our approach succeeded. A similar analysis applies to Mutex.

In the case of the tool Party, for the arb and farb examples Party was able to find solutions faster than exp2. In these cases, Party uses a *cut-off* of 4, i.e., it reduces configurations with $n > 4$ processes to the case $n = 4$. However, even though Party has several optimizations for token ring systems, our tool was able to synthesize an implementation of the parb, while Party timed out for this case.

Threats to validity. Our experimental evaluation is subject to some threats. The specification language of PSketch is different from ours; it takes as input incomplete programs, which the tool tries to complete to satisfy the (safety) specification, while our tool takes as input specifications in the pre/post-conditions style. To mitigate this, our PSketch specifications consist of a set of actions that follow the pre/post-condition style, which must be executed in a determined way to satisfy the global requirement. Reordering code execution is one of the main capabilities of PSketch, and so this is solvable with PSketch. On the other hand, Party takes as input a set of LTL formulas using assume/guarantee specifications; in this case, we used the same inputs for our tool.

Additional remarks. Our tool can find multiple solutions for the examples. For instance, in the phil example, after identifying an initial solution, the algorithm continued, and it was able to find a second solution. We note that synthesizing distributed programs is a challenging task, even for simple cases. To our knowledge, no other tool can synthesize distributed solutions for some of these examples, such as the generalized Peterson's algorithm with a liveness property.

8 Related Work

The seminal work of Emerson and Clarke [12] and related approaches [4, 13] are based on tableau methods for synthesizing synchronization skeletons from global CTL specifications, which are impractical in most cases, or require the description of processes as state machines. The synthesis of distributed reactive systems [26] aims to synthesize a distributed architecture that reacts to an environment when possible. This is a very general problem, and it is undecidable in many cases. Bounded synthesis [15] focuses on reactive (synchronous) systems, or assumes a *causal memory* model [14], i.e., the components memorize their histories and synchronize according to this. Additionally, we note that all these approaches consider only global LTL specifications; our approach also considers local specifications in a pre/post-condition and invariant style.

Counterexample-driven guided inductive synthesis (CEGIS) [1] reduces the problem of finding a given program to solve a (bounded) first-order formula. Inductive generalization and verifiers are used to generate candidate solutions. Alloy* [22] implements a general version of CEGIS. In contrast to CEGIS, our approach begins with "loose" versions of the processes, which are refined until a solution is obtained. Thus, we accumulate counterexamples and use them throughout the synthesis process. As noted in [27], the use of executions of asynchronous concurrent programs as counterexamples in CEGIS is not direct. In particular, PSketch can only handle safety properties. Other works, e.g., [19, 29], consider only safety properties or are tailored to specific settings.

9 Final Remarks

We presented a correct-by-construction bounded approach for synthesizing distributed algorithms. The approach iteratively generates candidate process implementations using the Alloy tool and employs counterexamples to refine the search. In contrast to other methods, our synthesis algorithm performs local reasoning to avoid explicitly constructing the global state space and applies to general temporal properties, including safety and liveness. We developed a prototype tool that effectively solved diverse examples. As noted in Section 7, the number of shared variables may affect the scalability of our algorithm. In view of this, we plan to extend our approach in several ways, e.g., exploring other heuristics to improve the exploration of the instance space. One benefit of the proposed approach is the versatility provided by the Alloy notation, which enables us to model various kinds of distributed architectures, such as the token ring architecture described in Section 7.

References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 270–288. doi:10.1007/978-3-319-96145-3_15

[2] ARM Ltd. [n. d.]. AMBA Specification Rev.2 (1999). Available at www.arm.com.

[3] Anish Arora and Mohamed G. Gouda. 1993. Closure and Convergence: A Foundation of Fault-Tolerant Computing. *IEEE Trans. Software Eng.* 19, 11 (1993), 1015–1027. doi:10.1109/32.256850

[4] Paul C. Attie. 2016. Synthesis of large dynamic concurrent programs from dynamic specifications. *Formal Methods Syst. Des.* 48, 1-2 (2016), 94–147. doi:10.1007/S10703-016-0252-9

[5] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking.* MIT Press.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1579)*, Rance Cleaveland (Ed.). Springer, 193–207. doi:10.1007/3-540-49059-0_14

[7] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. 2007. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*, Rudy Lauwereins and Jan Madsen (Eds.). EDA Consortium, San Jose, CA, USA, 1188–1193. https://dl.acm.org/citation.cfm?id=1266622

[8] Pablo F. Castro, Luciano Putruele, Nazareno Aguirre, and Renzo Degiovanni. 2025. *PhilStone: A Tool for Bounded Synthesis of Synchronized Distributed Models from Lightweight Specifications.* doi:10.6084/m9.figshare.30551120.v2

[9] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2404)*, Ed Brinksma and Kim Guldstrand Larsen (Eds.). Springer, 359–364. doi:10.1007/3-540-45657-0_29

[10] Edsger W. Dijkstra. 1971. Hierarchical Ordering of Sequential Processes. *Acta Informatica* 1 (1971), 115–138. doi:10.1007/BF00289519

[11] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. doi:10.1145/360933.360975

[12] E. Allen Emerson and Edmund M. Clarke. 1982. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Sci. Comput. Program.* 2, 3 (1982), 241–266. doi:10.1016/0167-6423(83)90017-5

[13] E. Allen Emerson and Roopsha Samanta. 2011. An Algorithmic Framework for Synthesis of Concurrent Programs. In *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6996)*, Tevfik Bultan and Pao-Ann Hsiung (Eds.). Springer, 522–530. doi:10.1007/978-3-642-24372-1_41

[14] Bernd Finkbeiner and Ernst-Rüdiger Olderog. 2017. Petri games: Synthesis of distributed systems with causal memory. *Inf. Comput.* 253 (2017), 181–203. doi:10.1016/J.IC.2016.07.006

[15] Bernd Finkbeiner and Sven Schewe. 2013. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 519–539. doi:10.1007/S10009-012-0228-Z

[16] Wan Fokkink. 2013. *Distributed Algorithms: An Intuitive Approach.* The MIT Press.

[17] Daniel Jackson. 2016. *Software Abstractions: Logic, Language and Analysis.* MIT Press.

[18] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *ICSE 2010* (Cape Town, South Africa). Association for Computing Machinery, New York, NY, USA, 10 pages. doi:10.1145/1806799.1806833

[19] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. 2013. PARTY Parameterized Synthesis of Token Rings. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 928–933. doi:10.1007/978-3-642-39799-8_66

[20] Zohar Manna and Pierre Wolper. 1984. Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 6, 1 (1984), 68–93. doi:10.1145/357233.357237

[21] John McCarthy and Patrick J. Hayes. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, B. Meltzer and D. Michie (Eds.). Edinburgh University Press, 463–502. reprinted in McC90.

[22] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. In *Proc. of 37th IEEE/ACM International Conference on Software Engineering ICSE 2015.* doi:10.1109/ICSE.2015.77

[23] Doron Peled and Thomas Wilke. 1997. Stutter-invariant temporal properties are expressible without the next-time operator. *Inform. Process. Lett.* 63, 5 (1997). doi:10.1016/S0020-0190(97)00133-6

[24] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of Reactive(1) Designs. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3855)*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). Springer, 364–380. doi:10.1007/11609773_24

[25] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of an Asynchronous Reactive Module. In *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 372)*, Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca (Eds.). Springer, 652–671. doi:10.1007/BFB0035790

[26] A. Pnueli and R. Rosner. 1990. Distributed Reactive Systems Are Hard to Synthesize. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (SFCS '90).* IEEE Computer Society, USA, 1 pages. doi:10.1109/FSCS.1990.89597

[27] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008.* doi:10.1145/1375581.1375599

[28] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. *SIGPLAN Not.* 45, 1 (Jan. 2010), 14 pages. doi:10.1145/1707801.1706337

[29] Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 327–338. doi:10.1145/1706299.1706338