Automatic Completion of Distributed Protocols with Symmetry[⋆]

Rajeev Alur[1], Mukund Raghothaman[1],
Christos Stergiou[1,2], Stavros Tripakis[2,3], and Abhishek Udupa[1]

[1] University of Pennsylvania, Philadelphia, USA
[2] University of California, Berkeley, USA
[3] Aalto University, Helsinki, Finland

Abstract. A distributed protocol is typically modeled as a set of communicating processes, where each process is described as an extended state machine along with fairness assumptions. Correctness is specified using safety and liveness requirements. Designing correct distributed protocols is a challenging task. Aimed at simplifying this task, we allow the designer to leave some of the guards and updates to state variables in the description of the protocol as unknown functions. The protocol completion problem then is to find interpretations for these unknown functions while guaranteeing correctness. In many distributed protocols, process behaviors are naturally symmetric, and thus, synthesized expressions are further required to obey symmetry constraints. Our counterexample-guided synthesis algorithm consists of repeatedly invoking two phases. In the first phase, candidates for unknown expressions are generated using the SMT solver Z3. This phase requires carefully orchestrating constraints to enforce the desired symmetry constraints. In the second phase, the resulting completed protocol is checked for correctness using a custom-built model checker that handles fairness assumptions, safety and liveness requirements, and exploits symmetry. When model checking fails, our tool examines a set of counterexamples to safety/liveness properties to generate constraints on unknown functions that must be satisfied by subsequent completions. For evaluation, we show that our prototype is able to automatically discover interesting missing details in distributed protocols for mutual exclusion, self stabilization, and cache coherence.

1 Introduction

Protocols for coordination among concurrent processes are an essential component of modern multiprocessor and distributed systems. The multitude of behaviors arising due to asynchrony and concurrency makes the design of such protocols difficult. Consequently, analyzing such protocols has been a central theme of

[⋆] This research was partially supported by NSF Expeditions award CCF 1138996 and by NSF award #1329759. The authors also acknowledge support from the Academy of Finland and the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies).

research in formal verification for decades. Now that verification tools are mature enough to be applied to find bugs in real-world protocols, a promising area of research is *protocol synthesis*, aimed at simplifying the design process via more intuitive programming abstractions to specify the desired behavior.

Traditionally, a distributed protocol is modeled as a set of communicating processes, where each process is described by an extended state machine. The correctness is specified by both safety and liveness requirements. In *reactive synthesis* [26,24,5], the goal is to automatically derive a protocol from its correctness requirements specified in temporal logic. However, if we require the implementation to be distributed, then reactive synthesis is undecidable [25,20,31,13]. An alternative, and potentially more feasible approach inspired by *program sketching* [28], is to ask the programmer to specify the protocol as a set of communicating state machines, but allow some of the guards and updates to state variables to be unknown functions, to be completed by the synthesizer so as to satisfy all the correctness requirements. This methodology for protocol specification can be viewed as a fruitful collaboration between the designer and the synthesis tool: the programmer has to describe the structure of the desired protocol, but some details that the programmer is unsure about, for instance, regarding corner cases and handling of unexpected messages, will be filled in automatically by the tool.

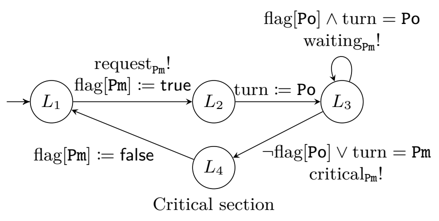
In our formalization of the synthesis problem, processes communicate using input/output channels that carry typed messages. Each process is described by a state machine with a set of typed state variables. Transitions consist of (1) guards that test input messages and state variables and, (2) updates to state variables and fields of messages to be sent. Such guards and updates can involve *unknown* (typed) functions to be filled in by the synthesizer. In many distributed protocols, such as cache coherence protocols, processes are expected to behave in a symmetric manner. Thus, we allow variables to have *symmetric types* that restrict the read/write accesses to obey symmetry constraints. To specify safety and liveness requirements, the state machines can be augmented with acceptance conditions that capture incorrect executions. Finally, fairness assumptions are added to restrict incorrect executions to those that are *fair*. It is worth noting that in verification one can get useful analysis results by focusing solely on safety requirements. In synthesis, however, ignoring liveness requirements and fairness assumptions, typically results in trivial solutions. The protocol completion problem, then, is, given a set of extended state machines with unknown guards and update functions, to find expressions for the unknown functions so that the composition of the resulting machines does not have an accepting fair execution.

Our synthesis algorithm relies on a counterexample-guided strategy with two interacting phases: candidate interpretations for unknown functions are generated using the SMT solver Z3 and the resulting completed protocol is verified using a model checker. We believe that our realization of this strategy leads to the following contributions. First, while searching for candidate interpretations for unknown functions, we need to generate constraints that enforce symmetry in an accurate manner without choking current SMT solvers. Second, surprisingly there is no publicly available model checker that handles all the features that we critically need, namely, symmetry, liveness requirements, and fairness assumptions. So, we had to develop our own model checker, building on the known theoretical foundations. Third, we develop an algorithm that examines the counterexamples to safety/liveness requirements when model checking fails, and generates constraints on unknown functions that must be satisfied in subsequent completions. Finally, the huge search space for candidate expressions is a challenge for the scalability for any synthesis approach. As reported in Section 5, we experimented with many alternative strategies for prioritizing the search for candidate expressions, and this experience offers some insights regarding what information a user can provide for getting useful results from the synthesis tool. We evaluate our synthesis tool in completing a mutual exclusion protocol, a self stabilization protocol and a non-trivial cache coherence protocol. Large parts of the behavior of the protocol were left unspecified in the case of the mutual exclusion protocol and the self stabilization protocol, whereas the cache coherence protocol had quite a few tricky details left unspecified. Our tool synthesized the correct completions for these protocols in a reasonable amount of time.

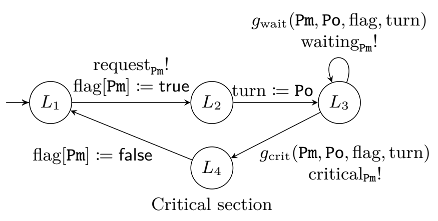
Related Work. *Bounded synthesis* [14] and *genetic programming* [18,19] are other approaches for handling the undecidability of distributed reactive synthesis. In the first, the size of the implementation is restricted, and in the second the implementation space is sampled and candidates are mutated in a stochastic process. The problem of inferring extended finite-state machines has been studied in the context of active learning [6]. The problem of completing distributed protocols has been targeted by the works presented in [2,32] and *program repair* [17] addresses a similar problem. Compared to [2], our algorithm can handle extended state machines that include variables and transitions with symbolic expressions as guards and updates. Compared to [32], our algorithm can also handle liveness violations and, more importantly, can process counterexamples automatically. PSKETCH [29] is an extension of the *program sketching* work for concurrent data structures but is limited to safety properties. The work in [15] describes an approach based on QBF solvers for synthesizing a distributed self-stabilizing system, which also approximates liveness with safety and uses templates for the synthesized functions. Also, compared to all works mentioned above, our algorithm can be used to enforce symmetry in the synthesized processes.

2 An Illustrative Example

Consider Peterson's mutual exclusion algorithm, described in Figure 1a, which manages two symmetric processes contending for access to a critical section. Each process is parameterized by `Pm` and `Po` (for "my" process id and "other" process id respectively), such that $\texttt{Pm} \neq \texttt{Po}$. Both parameters `Pm` and `Po` are of type processid and they are allowed to take on values `P0` and `P1`. We therefore have two instances: P_0, where $(\texttt{Pm} = \texttt{P0}, \texttt{Po} = \texttt{P1})$, and P_1, where $(\texttt{Pm} = \texttt{P1}, \texttt{Po} = \texttt{P0})$. P_0 and P_1 communicate through the shared variables *turn* and *flag*. The variable *turn* has type processid. The *flag* variable is an array of Boolean values, with index type processid. The objective of the protocol is to control access to the



(a) Parameterized Symmetric Process



(b) Incomplete process sketch

Fig. 1: Peterson's mutual exclusion algorithm. The non-trivial guards of the (L_3, L_3) and (L_3, L_4) transitions in Figure 1(a) have been replaced in Figure 1(b) by "unknown" functions g_{wait} and g_{crit} respectively.

critical section, represented by location L_4, and ensure that both of the processes P_0 and P_1 are never simultaneously in the critical section.

The liveness monitor shown in Figure 2 captures the requirement that a process does not wait indefinitely to enter the critical section. The monitor accepts all undesirable runs where a process has requested access to the critical section but never reaches state L_4 after. The messages request, waiting, and critical inform the liveness monitor about the state of the processes, and the synchronization model here is that of communicating I/O automata [21]. Note that a run accepted by the monitor may be *unfair* with respect to some processes. Enforcing *weak* process fairness on P_0 and P_1 is sufficient to rule out unfair executions, but not necessary. Enforcing weak fairness on the transitions between (L_2, L_3), (L_3, L_4) and (L_4, L_1) suffices.

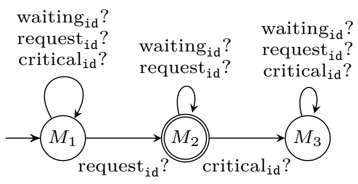


Fig. 2: Liveness Monitor

Now, suppose the protocol developer has trouble figuring out the exact condition under which a process is allowed to enter the critical section, but knows the structure of the processes P_0 and P_1, and requires them to be symmetric. Figure 1b describes what the developer knows about the protocol. The functions g_{wait} and g_{crit} represent unknown Boolean valued functions over the state variables and the parameters of the process under consideration. Including the parameters as part of the domain of g_{wait} and g_{crit} indicates that the completions for processes P_0 and P_1 need to be symmetric. The objective is to assist the developer by automatically discovering interpretations for these unknown functions, such that the completed protocol satisfies the necessary mutual exclusion property, and the requirements imposed by the liveness monitor. We formalize this completion problem in Section 3, and present our completion algorithm in Section 4.

3 Formalization

3.1 Extended State Machine Sketches

We model processes using Extended State Machine Sketches (ESM-S). Fix a collection of types, such as the type *bool* of the Boolean values {true, false}, enumerated types such as $\{\mathsf{red}, \mathsf{green}, \mathsf{blue}\}$, or finite subsets $nat[x, y]$ of natural numbers $\{i \mid x \leq i \leq y\}$. Other examples include *symmetric types* (described in Section 3.2), array and record types. Note that each type is required to be finite.

The description of an ESM-S will mention several function symbols. Some of these have interpretations which are already known, while others have unknown interpretations. Each function symbol, both known and unknown, is associated with a signature, $d_1 \times \cdots \times d_n \to r$, where $d_1, \ldots, d_n$ are the types of its arguments and r is the return type. Expressions may then be constructed using these function symbols, state variables, and input channels. Formally, an ESM-S A is a tuple $\langle L, l_0, I, O, S, \sigma_0, U, T, \mathcal{F}_s, \mathcal{F}_w\rangle$ such that:

– L is a finite set of locations and $l_0 \in L$ is the initial location,
– I and O are finite sets of typed input and output channels, respectively,
– S is a finite set of typed state variables,
– σ_0 maps each variable $x \in S$ to its initial value $\sigma_0(x)$,
– U is a set of unknown function symbols,
– T is a set of transitions of the form $\langle l,\ c,\ \mathsf{guard},\ \mathsf{updates},\ l'\rangle$, where $c \in I$, $c \in O$ and $c = \epsilon$ for input, output and internal transitions respectively. The transition is guarded by the expression guard and $\mathsf{updates}$ are the updates to state variables,
– $\mathcal{F}_s, \mathcal{F}_w \subseteq 2^{T_\epsilon \cup T_O}$, are sets of strong and weak fairnesses respectively. Here T_O and T_ϵ are the sets of output and internal transitions respectively.

A guard description guard is a Boolean expression over the state variables S that can use unknown functions from U. Similarly, an update description $\mathsf{updates}$ is a sequence of assignments of the form $\mathsf{lhs} := \mathsf{rhs}$ where lhs is one of the state variables or an output channel in the case of an output transition, and rhs is an expression over state variables or state variables and an input channel in the case of an input transition, possibly using unknown functions from U.

Executions. To define the executions of an ESM-S, we first pick an *interpretation* R which maps each unknown function $u \in U$ to an interpretation of u. Given a set of variables V, a *valuation* σ is a function which maps each variable $x \in V$ to a value $\sigma(x)$ of the corresponding type, and we write Σ_V for the set of all such valuations. Given a valuation $\sigma \in \Sigma_V$, a variable x, and a value v of appropriate type, we write $\sigma[x \mapsto v] \in \Sigma_{V \cup \{x\}}$ for the valuation which maps all variables $y \neq x$ to $\sigma(y)$, and maps x to v.

The executions of A are defined by describing the updates to the state valuation $\sigma \in \Sigma_S$ during each transition. Note that each guard description guard naturally defines a set $[\![\mathsf{guard}, R]\!]$ of valuations $\sigma \in \Sigma_S$ which satisfy guard with the unknown functions instantiated with R. Similarly, each update description $\mathsf{updates}$ defines a function $[\![\mathsf{updates}, R]\!]$ of type $\Sigma_{S \cup \{x\}} \to \Sigma_S$ for input transitions on the channel x, $\Sigma_S \to \Sigma_{S \cup \{y\}}$ for output transitions on the channel y, and $\Sigma_S \to \Sigma_S$ for internal transitions respectively. A *state* of an ESM-S A is a pair (l, σ) where, $l \in L$ and $\sigma \in \Sigma_S$. We then write:

– $(l, \sigma) \xrightarrow{x?v} (l', \sigma')$ if A has an input transition from l to l' on channel x with guard guard and update $\mathsf{updates}$ such that $\sigma \in [\![\mathsf{guard}, R]\!]$ and $[\![\mathsf{updates}, R]\!](\sigma[x \mapsto v]) = \sigma'$;

– $(l, \sigma) \xrightarrow{y!v} (l', \sigma')$ if A has an output transition from l to l' on channel y with guard guard and update $\mathsf{updates}$ such that $\sigma \in [\![\mathsf{guard}, R]\!]$ and $[\![\mathsf{updates}, R]\!](\sigma) = \sigma'[y \mapsto v]$; and
– $(l, \sigma) \xrightarrow{\epsilon} (l', \sigma')$ if A has an internal transition from l to l' with guard guard and update guard such that $\sigma \in [\![\mathsf{guard}, R]\!]$ and $[\![\mathsf{updates}, R]\!](\sigma) = \sigma'$.

We write $(l, \sigma) \to (l', \sigma')$ if either there are x, v such that $(l, \sigma) \xrightarrow{x?v} (l', \sigma')$, there are y, v such that $(l, \sigma) \xrightarrow{y!v} (l', \sigma')$, or $(l, \sigma) \xrightarrow{\epsilon} (l', \sigma')$. A finite (infinite) *execution* of the ESM-S A under R is then a finite (resp. infinite) sequence: $(l_0, \sigma_0) \to (l_1, \sigma_1) \to (l_2, \sigma_2) \to \cdots$ where for every $j \geq 0$, (l_j, σ_j) is a state of A, (l_0, σ_0) is an initial state of A, and for $j \geq 1$, $(l_j, \sigma_j) \to (l_{j+1}, \sigma_{j+1})$. A state (l, σ) is *reachable* under R if there exists a finite execution that reaches that state: $(l_0, \sigma_0) \to \cdots \to (l, \sigma)$. We say that a transition from l to l' with guard guard is *enabled* in state (l, σ) if $\sigma \in [\![\mathsf{guard}, R]\!]$. A state (l, σ) is called a *deadlock* if no transition is enabled in (l, σ). The ESM-S A is called deadlock-free under R if no deadlock state is reachable under R. The ESM-S A is called *deterministic* under R if for every state (l, σ), if there are multiple transitions enabled at (l, σ), then they must be input transitions on distinct input channels.

Consider a weak fairness requirement $F \in \mathcal{F}_w$. An infinite execution of A under R is called *fair* with respect to a weak fairness F if either: (a) for infinitely many indices i, none of the transitions $t \in F$ is enabled in (l_i, σ_i), or (b) for infinitely many indices j one of the transitions in F is taken at step j. Thus, for example, the necessary fairness assumptions for Peterson's algorithm are $\mathcal{F}_w = \{\{\tau_{23}\}, \{\tau_{34}\}, \{\tau_{41}\}\}$, where τ_{23}, τ_{34}, and τ_{41} refer to the (L_2, L_3), (L_3, L_4) and (L_4, L_1) transitions respectively. Similarly, an infinite execution of A under R is fair with respect to a strong fairness $F \in \mathcal{F}_s$ if either: (a) there exists k such that for every $i \geq k$ and every transition $t \in F$, t is not enabled in (l_i, σ_i), or (b) for infinitely many indices j one of the transitions in F is taken at step j. Finally, an infinite execution of A is fair if it is fair with respect to each strong and weak fairness requirement in $\mathcal{F}_s$ and $\mathcal{F}_w$ respectively.

Composition of ESM sketches. For lack of space, we only provide an informal definition of composition of ESM-S here. A formal definition can be found in the full version of this paper [3]. Informally, two ESM-S A_1 and A_2 are composed by synchronizing their output and input transitions on a given channel. If A_1 has an output transition on channel c from location l_1 to l_1' with guard and updates guard_1 and $\mathsf{updates}_1$, and A_2 has an input transition on the same channel c from location l_2 to l_2' with guard and updates guard_2 and $\mathsf{updates}_2$ then their product has an output transition from location (l_1, l_2) to (l_1', l_2') on channel c with guard $\mathsf{guard}_1 \wedge \mathsf{guard}_2$ and updates $\mathsf{updates}_1; \mathsf{updates}_2$. Note that by sequencing the updates, the value written to the channel c by A_1 is then used by subsequent updates of the variables of A_2 in $\mathsf{updates}_2$.

Specifications. An ESM-S can be equipped with error locations $L_e \subseteq L$, accepting locations $L_a \subseteq L$, or both. The composition of two ESM-S A_1, A_2 "inherits" the error and accepting locations of its components. A product location (l_1, l_2) is an error (accepting) location if either l_1 or l_2 are error (accepting) locations. An ESM-S A is called *safe* under R if for all reachable states (l, σ), l is not an error location. An infinite execution of A under R, $(l_0, \sigma_0) \to (l_1, \sigma_1) \to \cdots$, is called *accepting* if for infinitely many indices j, $l_j \in L_a$. A is called *live* under R if it has no infinite fair accepting executions.

3.2 Symmetry

It is often required that the processes of an ESM-S completion problem have some structurally similar behavior, as we saw in Section 2 in the case of Peterson's algorithm. To describe such requirements, we use *symmetric types*, which are similar to *scalarsets* used in the Murφ model checker [23].

A symmetric type T is characterized by: (a) its name, and (b) its cardinality $|T|$, which is a finite number. Given a collection of processes parameterized by a symmetric type T, such as P_0 and P_1 of Peterson's algorithm, the idea is that the system is invariant under permutations (i.e. renaming) of the parameter values. Let $\mathsf{perm}(T)$ be the set of all permutations $\pi_T : T \to T$ over the symmetric type T. For ease of notation, we define $\pi_T(v) = v$, for values v whose type is *not* T. Given the collection of all symmetric types $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ of the system, we can then describe permutations over $\mathcal{T}$ as the composition of permutations over the individual types, $\pi_{T_1} \circ \pi_{T_2} \circ \cdots \circ \pi_{T_n}$. Let $\mathsf{perm}(\mathcal{T})$ be the set of such "system-wide" permutations over $\mathcal{T}$.

ESM sketches and input and output channels may thus be parameterized by symmetric values. The state variables and array variable indices of an ESM-S may also be of symmetric type. Given the symmetric types $\mathcal{T}$ and an interpretation R of the unknown functions in an ESM-S A, we say that A is *symmetric* with respect to $\mathcal{T}$ if every execution $(l_0, \sigma_0) \to (l_1, \sigma_1) \to \cdots \to (l_n, \sigma_n) \to \cdots$ of A under R also implies the existence of the permuted execution $(\pi(l_0), \pi(\sigma_0)) \to (\pi(l_1), \pi(\sigma_0)) \to \cdots (\pi(l_n), \pi(\sigma_n)) \to \cdots$ of A, where the channel identifiers along transitions are also suitably permuted, for every permutation $\pi \in \mathsf{perm}(\mathcal{T})$.

We therefore require that any interpretation R considered be such that the completed ESM-S A is symmetric with respect to $\mathcal{T}$ under R. For every unknown function f in A, requiring that $\forall d \in \mathsf{dom}(f), \pi(f(d)) = f(\pi(d)))$, for each permutation $\pi \in \mathsf{perm}(\mathcal{T})$, ensures that the behavior of f is symmetric. In Section 4, we will describe how these additional constraints are presented to the SMT solver. Note that while we have only discussed *full symmetry* here, other notions of symmetry such as *ring symmetry* and *virtual symmetry* [11] can also be accommodated in our formalization.

3.3 Completion Problem

In many cases, the designer has some prior knowledge about the unknown functions used in an ESM-S. For example, the designer may know that the variable *turn* is read-only during the (L_3, L_4) transition of Peterson's algorithm. The designer may also know that the unknown guard of a transition is independent of some state variable. Many instances of such "prior knowledge" can already be expressed using the formalism just described: the update expression of *turn* in the unknown transition can be set to the identity function (in the first case), and the

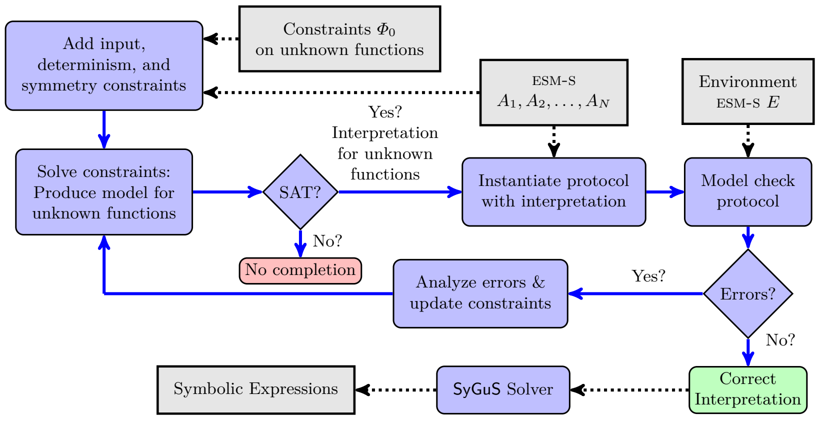

Constraints Φ_0 on unknown functions
Add input, determinism, and symmetry constraints
ESM-S $A_1, A_2, \ldots, A_N$
Environment ESM-S E
Yes? Interpretation for unknown functions
Solve constraints: Produce model for unknown functions
SAT?
Instantiate protocol with interpretation
Model check protocol
No?
No completion
Analyze errors & update constraints
Yes?
Errors?
No?
Symbolic Expressions
SyGuS Solver
Correct Interpretation

Fig. 3: Completion Algorithm.

designer can omit the irrelevant variable from the signature of the update function (in the second case). We also allow the designer to specify additional constraints on the unknown functions: she may know, as in the case of Peterson's algorithm for example, that $g_{\text{crit}}(\texttt{Pm}, \texttt{Po}, \mathit{flag}, \mathit{turn}) \vee g_{\text{wait}}(\texttt{Pm}, \texttt{Po}, \mathit{flag}, \mathit{turn})$, for every valuation of the function arguments Pm, Po, *flag*, and *turn*. This additional knowledge, which is helpful to guide the synthesizer, is encoded in the initial constraints Φ_0 imposed on candidate interpretations of U. Note that these constraints might refer to multiple unknown functions from the same or different ESM-S.

Formally, we can now state the completion problem as: Given a set of ESM-S $A_1, \ldots A_N$ with sets of unknown functions $U_1, \ldots, U_N$, an environment ESM-S E with an empty set of unknown functions, and a set of constraints Φ_0 on the unknown functions $U = U_1 \cup \cdots \cup U_N$, find an interpretation R of U, such that (a) $A_1, \ldots, A_N$ are deterministic under R, (b) the completed system $\Pi = A_1 \mid \cdots \mid A_N \mid E$ is symmetric with respect to $\mathcal{T}$ under R, where $\mathcal{T}$ is the set of symmetric types in the system, (c) R satisfies the constraints in Φ_0, and (d) the product Π under R is deadlock-free, safe, and live.

4 Solving the Completion Problem

The synthesis algorithm is outlined in Figure 3. We maintain a set of constraints Φ on possible completions, and repeatedly query Z3 [22] for candidate interpretations satisfying all constraints in Φ. If the interpretation is certified correct by the model checker, we are done. Otherwise, counter-example executions returned by the model checker are analyzed, and Φ is strengthened with further constraints which eliminate all subsequent interpretations with similar erroneous executions. If a symbolic expression is required, we can submit the correct interpretation to a SyGuS solver [1]. A SyGuS solver takes a set of constraints $\mathcal{C}$ on an unknown function f together with the search space for the body of f — expressed as a grammar — and finds an expression in the grammar for f, such that it satisfies the constraints $\mathcal{C}$. In this section, we first describe the initial determinism and symmetry constraints expected of all completions. Next, we briefly describe the model checker used in our implementation, and then describe how to analyze counterexamples returned by the model checker. Finally, we describe additional heuristics to bias the SMT solver towards intuitively simpler completions first.

4.1 Initial Constraints

Determinism Constraints. Recall that an ESM-S is deterministic under an interpretation R if and only if for every state (l, σ) if there are multiple transitions enabled at (l, σ), then they must be input transitions on distinct input channels. We constrain the interpretations chosen at every step such that all ESM-S in the protocol are deterministic. Consider the ESM-S for Peterson's algorithm shown in Figure 1b. We have two transitions from the location L_3, with guards $g_{\text{crit}}(\texttt{Pm}, \texttt{Po}, \mathit{flag}, \mathit{turn})$ and $g_{\text{wait}}(\texttt{Pm}, \texttt{Po}, \mathit{flag}, \mathit{turn})$. We ensure that these expressions never evaluate to true simultaneously with the constraint $\neg\exists v_1 v_2 v_3 v_4\,(g_{\text{crit}}(v_1, v_2, v_3, v_4) \wedge g_{\text{wait}}(v_1, v_2, v_3, v_4))$. Although this is a quantified expression, which can be difficult for SMT solvers to solve, note that we only support finite types, whose domains are often quite small. So our tool unrolls the quantifiers and presents only quantifier-free formulas to the SMT solver.

Symmetry Constraints. Suppose that the interpretation chosen for the guard g_{crit} shown in Figure 1b, was such that $g_{\text{crit}}(\texttt{P0}, \texttt{P1}, \langle\bot, \top\rangle, \texttt{P0}) = \texttt{true}$. Then for the ESM-S to be symmetric under this interpretation, we require that $g_{\text{crit}}(\texttt{P1}, \texttt{P0}, \langle\top, \bot\rangle, \texttt{P1} = \texttt{true}$ as well, because the latter expression is obtained by applying the permutation $\{\texttt{P0} \mapsto \texttt{P1}, \texttt{P1} \mapsto \texttt{P0}\}$ on the former expression. Note that the elements of the *flag* array in the preceding example were flipped, because *flag* is an array indexed by the symmetric type $\mathsf{processid}$. In general, given a function $f \in U_i$, we enforce the constraint $\forall \pi \in \mathsf{perm}(\mathcal{T}) \forall d \in \mathsf{dom}(f)(f(\pi(d)) \equiv \pi(f(d)))$, where $\mathcal{T}$ is the set of symmetric types that appear in A_i. As in the case of determinism constraints, we unroll the quantifiers here as well.

4.2 Model Checker

To effectively and repeatedly generate constraints to drive the synthesis loop, a model checker needs to: (a) support checking liveness properties, with algorithmic support for fine grained notions of strong and weak fairness, (b) dynamically prioritize certain paths over others (*cf.* Section 4.4), and (c) exploit symmetries inherent in the model. The fine grained notions of fairness over sets of transitions, rather than bulk process fairness are crucial. For instance, in the case of unordered channel processes, we often require that no message be delayed indefinitely, which cannot be captured by enforcing fairness at the level of the entire process. The ability to prioritize certain paths over others is also crucial so that candidate interpretations are exercised to the extent possible in one model checking run (*cf.* Section 4.4). Finally, support for symmetry-based state space reductions, while not absolutely crucial, can greatly speed up each model checking run.

Surprisingly, we found that none of the well-supported model checkers met all of our requirements. SPIN [16] only supports weak process fairness at an algorithmic level and does not employ symmetry-based reductions. Support for symmetry-based reductions is present in Murφ [23,10], but it lacks support for liveness checking. SMC [27] is a model checker with support for symmetry reduction and strong and weak process fairness. Unfortunately, it is no longer maintained, and has very rudimentary counterexample generation capabilities. Finally, NuSMV [8] does not support symmetry reductions, but supports strong and weak process level fairness. But bugs in the implementation of counterexample generation, left us unable to obtain counterexamples in some cases.

We therefore implemented a model checker based on the ideas used in Murφ [10] for symmetry reduction, and an adaptation of the techniques presented in earlier literature [12] for checking liveness properties under fairness assumptions. The model checking algorithm consists of the following steps: (1) construct the symmetry-reduced state graph, (2) find accepting strongly connected components (SCCs) in the reduced state graph, (3) delete unfair states from each SCC; repeat steps (2) and (3) until either a fair SCC is found or no more accepting SCCs remain. A more detailed description of the model checking algorithm is presented in the full version of the paper [3].

4.3 Analysis of Counterexamples

We now describe our algorithms for analyzing counterexamples by way of examples. A more formal description of the algorithms can be found in the full version of this paper [3].

Analyzing deadlocks. In Figure 1b, consider the candidate interpretation where both g_{crit}, g_{wait} are set to be universally false. Two deadlock states are then reachable: $S_1 = ((L_3, L_3), \{\mathit{flag} \mapsto \langle\top,\top\rangle, \mathit{turn} \mapsto \mathtt{P1}\}$ and $S_2 = ((L_3, L_3), \{\mathit{flag} \mapsto \langle\top,\top\rangle, \mathit{turn} \mapsto \mathtt{P0}\}$. We strengthen Φ by asserting that these deadlocks do not occur in future interpretations: either S_1 is unreachable, or the system can make a transition from S_1 (and similarly for S_2). In this example, the reachability of both deadlock states is not dependent on the interpretation, *i.e.*, the execution that leads to the states does not exercise any unknown function, hence, we need to make sure that the states are not deadlocks. The possible transitions out of location (L_3, L_3) are the transitions from L_3 to L_3 (waiting transition) and from L_3 to L_4 (critical transition) for each of the two processes. In each deadlock state, at least one of the four guards has to be true: $g_{\text{wait}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P1}) \vee g_{\text{crit}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P1}) \vee g_{\text{wait}}(\mathtt{P1}, \mathtt{P0}, \langle\top,\top\rangle, \mathtt{P1}) \vee g_{\text{crit}}(\mathtt{P1}, \mathtt{P0}, \langle\top,\top\rangle, \mathtt{P1})$ for S_1, and $g_{\text{wait}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P0}) \vee g_{\text{crit}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P0}) \vee g_{\text{wait}}(\mathtt{P1}, \mathtt{P0}, \langle\top,\top\rangle, \mathtt{P0}) \vee g_{\text{crit}}(\mathtt{P1}, \mathtt{P0}, \langle\top,\top\rangle, \mathtt{P0})$ for S_2. The two disjuncts are added to the set of constraints, since any candidate interpretation has to satisfy them in order for the resulting product to be deadlock-free.

Analyzing safety violations. Consider now an erroneous interpretation where the critical transition guards are $\mathtt{true}$ for both processes when *turn* is $\mathtt{P0}$, that is: $g_{\text{crit}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P0})$ and $g_{\text{crit}}(\mathtt{P1}, \mathtt{P0}, \langle\top,\top\rangle, \mathtt{P0})$ are set to $\mathtt{true}$. Under this interpretation the product can reach the error location (L_4, L_4). We perform a weakest precondition analysis on the corresponding execution to obtain a necessary condition under which the safety violation is possible. In this case, the execution crosses both critical transitions and the generated constraint is $\neg g_{\mathrm{crit}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P0}) \vee \neg g_{\mathrm{crit}}(\mathtt{P1}, \mathtt{P0}, \langle\top,\top\rangle, \mathtt{P0})$. Note that the conditions obtained from this analysis are necessary; the product under any interpretation that does not satisfy them will exhibit the same safety violation.

Analyzing liveness violations. An interpretation that satisfies the constraints gathered above is one that, when *turn* is $\mathtt{P0}$, enables both waiting transitions and disables the critical ones. Intuitively, under this interpretation, the two processes will not make progress if *turn* is $\mathtt{P0}$ when they reach L_3. The executions in which the processes are at L_3 and either P_0 or P_1 continuously take the waiting transition is an accepting one. As with safety violations, we eliminate liveness violations by adding constraints generated through weakest precondition analysis of the accepting executions. In this case, this results in two constraints: $\neg g_{\mathrm{wait}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P0})$ and $\neg g_{\mathrm{wait}}(\mathtt{P1}, \mathtt{P0}, \langle\top,\top\rangle, \mathtt{P0})$. However, in the presence of fairness assumptions, these constraints are too strong. This is because removing an execution that causes a fair liveness violation is not the only way to resolve it: another way is to make it unfair. Given the weak fairness assumption on the transitions on the $\mathrm{critical}_{\mathtt{Pi}}$ channels, the correct constraint generated for the liveness violation of Process P_0 is: $\neg g_{\mathrm{wait}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P0}) \vee g_{\mathrm{crit}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P0}) \vee g_{\mathrm{crit}}(\mathtt{P1}, \mathtt{P0}, \mathsf{true}, \mathsf{true}, \mathtt{P0})$, where the last two disjuncts render the accepting execution unfair.

4.4 Optimizations and Heuristics.

We describe a few key optimizations and heuristics that improve the scalability and predictability of our technique.

Not all counterexamples are created equal. The constraint we get from a single counter-example trace is weaker when it exercises a large number of unknown functions. Consider, for example, a candidate interpretation for the incomplete Peterson's algorithm which, when $turn = \mathtt{P0}$, sets both waiting transition guards g_{wait} to true, and both critical transition guards g_{crit} to false. We have already seen that the product is not live under this interpretation. From the infinite execution leading up-to the location (L_3, L_3), and after which P_0 loops in L_3, we obtain the constraint $\neg g_{\mathrm{wait}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P0})$. On the other hand, if we had considered the longer self-loop at (L_3, L_3), where P_0 and P_1 alternate in making waiting transitions, we would have obtained the weaker constraint $\neg g_{\mathrm{wait}}(\mathtt{P0}, \mathtt{P1}, \langle\top,\top\rangle, \mathtt{P0}) \vee \neg g_{\mathrm{wait}}(\mathtt{P1}, \mathtt{P0}, \langle\top,\top\rangle, \mathtt{P0})$. In general, erroneous traces which exercise fewer unknown functions have the potential to prune away a larger fraction of the search space and are therefore preferable over traces exercising a larger number of unknown functions.

In each iteration, the model checker discovers several erroneous states. In the event that the candidate interpretation chosen is blatantly incorrect, it is infeasible to analyze paths to all error states. A naïve solution would be to analyze paths to the first n errors states discovered (where n is configurable). But depending on the strategy used to explore the state space, a large fraction these errors could be similar, and would only provide us with rather weak constraints. On the other hand, exercising as many unknown functions as possible, along different paths, has the potential to provide stronger constraints on future interpretations. In summary, we bias the model checker to *cover* as many unknown functions as possible, such that each path exercises as few unknown functions as possible.

Heuristics/Prioritizations to guide the SMT solver. As mentioned earlier, we use an SMT solver to obtain interpretations for unknown functions, given a set of constraints. When this set is small, as is the case at the beginning of the algorithm, there exist many satisfying interpretations. At this point the interpretation chosen by the SMT solver can either lead the rest of the search down a "good" path, or lead it down a futile path. Therefore the run time of the synthesis algorithm can depend heavily on the interpretations returned by the SMT solver, which we consider a non-deterministic black box in our approach.

To reduce the influence of non-determinism of the SMT solver on the run time of our algorithm, we bias the solver towards specific forms of interpretations by asserting additional constraints. These constraints associate a *cost* with interpretations and require an interpretation with a given bound on the cost, which is relaxed whenever the SMT solver fails to find a solution.

We briefly describe the most important of the heuristics/prioritization techniques: (1) We minimize the number of points in the domain of an unknown guard function at which it evaluates to true. This results in minimally permissive guards. (2) Based on the observation that most variables are unchanged in a given transition, we prioritize interpretations where update functions leave the value of the variable unchanged, as far as possible. (3) We can also try to minimize the number of arguments on which the value of an unknown function depends.

5 Experiments

5.1 Peterson's Mutual Exclusion Protocol

In addition to the missing guards g_{grit} and g_{wait}, we also replace the update expressions of flag[`Pm`] in the (L_1, L_2) and (L_4, L_1) transitions with unknown functions that depend on all state variables. In the initial constraints we require that $g_{\text{crit}}(\texttt{Pm}, \texttt{Po}, \mathit{flag}, \mathit{turn}) \vee g_{\text{wait}}(\texttt{Pm}, \texttt{Po}, \mathit{flag}, \mathit{turn})$. The synthesis algorithm returns with an interpretation in less than a second. Upon submitting the interpretation to a SyGuS solver, the synthesized expressions match the ones shown in Figure 1b.

5.2 Self-stabilizing Systems

Our next case study is the synthesis of self-stabilizing systems [9]. A distributed system is self-stabilizing if, starting from an arbitrary initial state, in each execution, the system eventually reaches a global *legitimate* state, and only legitimate states are ever visited after. We also require that every legitimate state be reachable from every other legitimate state. Consider N processes connected in a line. Each process maintains two Boolean state variables `x` and `up`. The processes are described using guarded commands of the form, "`if` *guard* `then` *update*". Whether a command is enabled is a function of the variable values `x` and `up` of the process itself, and those of its neighbors. We attempted to synthesize the guards and updates for the middle two processes of a four process system P_1, P_2, P_3, P_4. Specifically, the ESM-S for P_2 and P_3 have two transitions, each with an unknown function as a guard and two unknown functions for updating its state variables. The guard is a function of x_{i-1}, x_i, x_{i+1}, up_{i-1}, up_i, up_{i+1}, and the updates of x_i and up_i are functions of x_i and up_i. We followed the definition in [15] and defined a state as being legitimate if exactly one guarded command is enabled globally. We also constrain the completions of P_2 and P_3 to be identical.

Due to the large number of unknown functions needed to be synthesized in this experiment and, in particular, because there were a lot of input domain points at which the guards had to be true, the heuristic that prefers minimally permissive guards, described in Section 4, was not effective. However, the heuristic that prioritizes interpretations in which the guards depend on fewer arguments of their domain was effective. For state variable updates, we applied the heuristic that prioritizes functions that leave the state unchanged or set it to a constant. After passing the synthesized interpretation through a SyGuS solver, the expressions we got were exactly the same as the ones found in [9].

5.3 Cache Coherence Protocol

A cache coherence protocol ensures that the copies of shared data in the private caches of a multiprocessor system are kept up-to-date with the most recent version. We describe the working of the German cache coherence protocol, which is often used as a case study in model checking research [7,30]. The protocol consists of a *Directory* process, n symmetric *Cache* processes and n symmetric *Environment* processes, one for each cache process. Each cache may be in the E, S or I state, indicating read-write, read, and no permissions on the data respectively. All communication between the caches and the directory is non-blocking, and occurs over buffered, unordered communication channels.

The environment issues *read* and *write* commands to its cache. In response to a *read* command, the cache C sends a *RequestS* command to the directory. The directory, sends C the most up-to-date copy of the data, after coordinating with other caches, and grants read access to C, and remembers C as a *sharer* of the data. In response to a *write* request from the environment, the cache C sends a *RequestE* command to the directory. The directory coordinates with every other cache C' that has read or write permissions to revoke their permissions and then grants C exclusive access to the data, and remembers C as the owner of the data. The complete German/MSI protocol, modeled as communicating extended state machines, is fairly complex, with a symmetry-reduced state space of about 20,000 states when instantiated with two cache processes and about 450,000 states when instantiated with three cache processes.

We consider a more complex variant of the German cache coherence protocol to evaluate the techniques we have presented so far, which we refer to as German/MSI. The main differences from the base German protocol are: (1) Direct communication between caches is possible in some cases, (2) A cache in the S state can silently relinquish its permissions, which can cause the directory to have out-of-date information about the caches which are in the S state. (3) A cache in the E state can coordinate with the directory to relinquish their permissions. A complete list of scenarios typically used when describing this protocol is presented

Benchmark	# UF	Search Space	# States	# Iters.	SMT Time (s)	Total Time (s)
Peterson	3	2^{36}	60	14	0.1	0.13
Dijkstra	6	2^{192}	~2000	30	27	64
German/MSI-2	16	$\sim 2^{4700}$	~20000 (symm. red.)	217	31	298
German/MSI-4	28	$\sim 2^{7614}$	~20000 (symm. red.)	419	898	1545
German/MSI-5	34	$\sim 2^{9000}$	~20000 (symm. red.)	525	2261	3410

Table 1: Experimental Results

in the full version of the paper [3]. These scenarios however, do not describe the protocol's behavior in several cases induced by concurrency.

Consider the scenario shown in Figure 4, where initially, cache C1 is in the I state, in contrast, the directory records that C1 is in state S and is a sharer, due to C1 having silently relinquished its read permissions at some point in the past. Now, both caches C1 and C2 receive *write* commands from their respective environments. Cache C2 sends a *RequestE* message to the directory, requesting exclusive write permissions. The directory, under the impression that C1 is in state S, sends an *Inv* message to it, informing it that C2 has requested exclusive access and C1 needs to acknowledge that it has relinquished permissions to C2. Concurrently, cache C1 sends a *RequestE* message to the directory requesting write permissions as well, which gets delayed. Subsequently, the cache C1 receives an invalidation when it is in the state IM, which cannot happen in the base German protocol. The correct behavior for the cache in this situation (shown by dashed arrows), is to send an *InvAck* message to the cache C2. The guard, the state variable updates, as well as the location update is what we have left unspecified in the case of this particular scenario. As part of the evaluation, we successfully synthesized the behavior of the German/MSI protocol in five such corner-case scenarios arising from concurrency. A description of the other corner-case scenarios can be found in the full version of the paper [3].

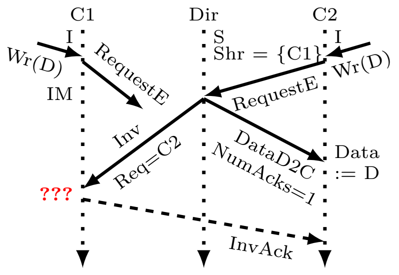


Fig. 4: A Racy Scenario

5.4 Summary of Experimental Results

Table 1 summarizes our experimental findings. All experiments were performed on a Linux desktop, with an Intel Core i7 CPU running at 3.4 GHz., with 8 GB of memory. The columns show the name of the benchmark, the number of unknown functions that were synthesized (# UF), the size of the search space for the unknown functions, the number of states in the complete protocol (# States), "symm. red." denotes symmetry reduced state space. The "# Iters." column shows the number of algorithm iterations, while the last two columns show the total amount of time spent in SMT solving and the end-to-end synthesis time.

The "German/MSI-n" rows correspond to the synthesizing the unknown behavior for the German/MSI protocol, with n out of the five unknown transitions left unspecified. In each case, we applied the heuristic to obtain minimally permissive guards and biased the search towards updates which leave the values of state variables unchanged as far as possible, except in the case of the Dijkstra benchmark, as mentioned in Section 5.2. Also, note that we ran each benchmark multiple times with different random seeds to the SMT solver, and report the worst of the run times in Table 1.

Programmer Assistance. In all cases, the programmer specified the kinds of messages to handle in the states where the behavior was unknown. For example, in the case of the German/MSI protocol, the programmer indicated that in the IM state on the cache, it needs to handle an invalidation from the directory (see Figure 4). In general, the programmer specified *what* needs to be handled, but not the *how*. This was crucial to getting our approach to scale.

Overhead of Decision Procedures. We observe from Table 1 that for the longer running benchmarks, the run time is dominated by SMT solving. In all of these cases, a very large fraction of the constraints asserted into the SMT solver are constraints to implement heuristics which are specifically aimed at guiding the SMT solver, and reducing the impact of non-deterministic choices made by the solver. Specialized decision procedures that handle these constraints at an algorithmic level [4] can greatly speed up the synthesis procedure.

Synthesizing Symbolic Expressions. The interpretations returned by the SMT solver are in the form of tables, which specify the output of the unknown function on specific inputs. We mentioned that if a symbolic expression is required we can pass this output to a SyGuS solver, which will then return a symbolic expression. We were able to synthesize compact expressions in all cases using the enumerative SyGuS solver [1]. Further, although the interpretations are only guaranteed to be correct for the finite instance of the protocol, the symbolic expressions generated by the SyGuS solver were *parametric*. We found that they were general enough to handle larger instances of protocol.

6 Conclusions

We have presented an algorithm to complete symmetric distributed protocols specified as ESM sketches, such that they satisfy the given safety and liveness properties. A prototype implementation, which included a custom model checker, successfully synthesized non-trivial portions of Peterson's mutual exclusion protocol, Dijkstra's self-stabilizing system, and the German/MSI cache coherence protocol. We show that programmer assistance in the form of *what* needs to be handled is crucial to the scalability of the approach. Scalability is currently limited by the scalability of the SMT solver. As part of future work, we plan to investigate algorithms that do not depend as heavily on SMT solvers as a core decision procedure. We are hopeful that such an approach will improve the scalability of our algorithms.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided Synthesis. In: FMCAD. pp. 1–17 (2013)
2. Alur, R., Martin, M.M.K., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Synthesizing Finite-State Protocols from Scenarios and Requirements. In: Yahav, E. (ed.) 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8855, pp. 75–91. Springer (2014)
3. Alur, R., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Automatic Completion of Distributed Protocols with Symmetry. CoRR arXiv:1505.04409 (2015), `http://arxiv.org/abs/1505.04409`
4. Bjorner, N., Phan, A.D.: νZ - Maximal Satisfaction with Z3. In: Kutsia, T., Voronkov, A. (eds.) SCSS 2014. EPiC Series, vol. 30, pp. 1–9. EasyChair (2014)
5. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. J. Comput. Syst. Sci. 78(3) (2012)
6. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Learning Extended Finite State Machines. In: Giannakopoulou, D., SalaÃijn, G. (eds.) Software Engineering and Formal Methods, Lecture Notes in Computer Science, vol. 8702, pp. 250–264. Springer International Publishing (2014)
7. Chou, C.T., Mannava, P., Park, S.: A Simple Method for Parameterized Verification of Cache Coherence Protocols. In: Hu, A., Martin, A. (eds.) Formal Methods in Computer-Aided Design. Lecture Notes in Computer Science, vol. 3312, pp. 382–398. Springer Berlin Heidelberg (2004)
8. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An Open-source Tool for Symbolic Model Checking. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer Berlin Heidelberg (2002)
9. Dijkstra, E.W.: Self-stabilizing Systems in Spite of Distributed Control. Commun. ACM 17(11), 643–644 (Nov 1974)
10. Dill, D.L.: The Murφ Verification System. In: Proceedings of the 8th International Conference on Computer Aided Verification. pp. 390–393. CAV '96, Springer-Verlag, London, UK, UK (1996)
11. Emerson, E.A., Havlicek, J.W., Trefler, R.J.: Virtual Symmetry Reduction. In: Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2000). pp. 121–131 (June 2000)
12. Emerson, E.A., Sistla, A.P.: Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. ACM Trans. Program. Lang. Syst. 19(4), 617–638 (1997)
13. Finkbeiner, B., Schewe, S.: Uniform Distributed Synthesis. In: IEEE Symposium on Logic in Computer Science. pp. 321–330 (2005)
14. Finkbeiner, B., Schewe, S.: Bounded synthesis. Software Tools for Tchnology Transfer 15(5-6), 519–539 (2013)
15. Gascón, A., Tiwari, A.: Synthesis of a Simple Self-stabilizing System. In: Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014. pp. 5–16 (2014)
16. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. Softw. Eng. 23(5), 279–295 (May 1997)

17. Jobstmann, B., Griesmayer, A., Bloem, R.: Program Repair as a Game. In: Computer Aided Verification, 17th International Conference. pp. 226–238. LNCS 3576 (2005)
18. Katz, G., Peled, D.: Model Checking-Based Genetic Programming with an Application to Mutual Exclusion. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference. pp. 141–156. LNCS 4963 (2008)
19. Katz, G., Peled, D.: Synthesizing Solutions to the Leader Election Problem Using Model Checking and Genetic Programming. In: Haifa Verification Conference. pp. 117–132 (2009)
20. Lamouchi, H., Thistle, J.: Effective Control Synthesis for DES Under Partial Observations. In: 39th IEEE Conference on Decision and Control. pp. 22–28 (2000)
21. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
22. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer Berlin Heidelberg (2008)
23. Norris IP, C., Dill, D.: Better Verification through Symmetry. Formal Methods in System Design 9(1-2), 41–75 (1996)
24. Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module. In: Proceedings of the 16th ACM Symposium on Principles of Programming Languages (1989)
25. Pnueli, A., Rosner, R.: Distributed Reactive Systems Are Hard to Synthesize. In: 31st Annual Symposium on Foundations of Computer Science. pp. 746–757 (1990)
26. Ramadge, P., Wonham, W.: The Control of Discrete Event Systems. IEEE Transactions on Control Theory 77, 81–98 (1989)
27. Sistla, A.P., Gyuris, V., Emerson, E.A.: SMC: A Symmetry-based Model Checker for Verification of Safety and Liveness Properties. ACM Trans. Softw. Eng. Methodol. 9(2), 133–166 (2000)
28. Solar-Lezama, A., Rabbah, R., Bodik, R., Ebcioglu, K.: Programming by Sketching for Bit-streaming Programs. In: Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation (2005)
29. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching Concurrent Data Structures. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '08 (2008)
30. Talupur, M., Tuttle, M.R.: Going with the Flow: Parameterized Verification Using Message Flows. In: Cimatti, A., Jones, R.B. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008. pp. 1–8. IEEE (2008)
31. Tripakis, S.: Undecidable Problems of Decentralized Observation and Control on Regular Languages. Information Processing Letters 90(1), 21–28 (Apr 2004)
32. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M., Alur, R.: TRANSIT: Specifying Protocols with Concolic Snippets. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 287–296. PLDI '13 (2013)