Symmetric Synthesis*†

Rüdiger Ehlers[1] and Bernd Finkbeiner[2]

1 **University of Bremen, Bremen, Germany**
`ruediger.ehlers@uni-bremen.de`
2 **Saarland University, Saarbrücken, Germany**
`finkbeiner@cs.uni-saarland.de`

Abstract

We study the problem of determining whether a given temporal specification can be implemented by a symmetric system, i.e., a system composed from identical components. Symmetry is an important goal in the design of distributed systems, because systems that are composed from identical components are easier to build and maintain. We show that for the class of rotation-symmetric architectures, i.e., multi-process architectures where all processes have access to all system inputs, but see different rotations of the inputs, the symmetric synthesis problem is EXPTIME-complete in the number of processes. In architectures where the processes do not have access to all input variables, the symmetric synthesis problem becomes undecidable, even in cases where the standard distributed synthesis problem is decidable.

1998 ACM Subject Classification D.2.4 Formal Methods

Keywords and phrases Reactive Synthesis, Symmetry

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2017.26

1 Introduction

Many classical protocols and distributed systems are *symmetric*. This means that every process, independently of its identity, starts in the same initial state and follows the same set of transitions. Symmetric systems are easier to understand and maintain; especially in VLSI designs, which usually contain large numbers of identical components, this is a significant cost factor. Constructing symmetric systems is also a step towards building arbitrarily scalable systems [8, 2, 11].

There is a large body of results [1, 18, 5, 12, 26, 13] that deal with the question of which distributed systems need symmetry breaking and which do not. *Leader election* among the processes on a ring, for example, cannot be implemented symmetrically [1]; similarly, in resource-sharing problems, like the *Dining Philosophers*, the only way to avoid starvation is to break the symmetry [18].

Our goal is to automate this type of reasoning. Given a specification of a reactive system in temporal logic, we wish to automatically determine whether there exists a symmetric implementation. This is a refinement of the classic *distributed synthesis problem*, which asks whether a temporal specification has an implementation where the processes are arranged

* This work was partially supported by the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative, by the German Research Foundation (DFG) within the program "Performance Guarantees for Computer Systems" and the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), and by the European Research Council (ERC) Grant OSARES (No. 683300).

† A full version of the paper is available at [7], `https://arxiv.org/abs/1710.05633`

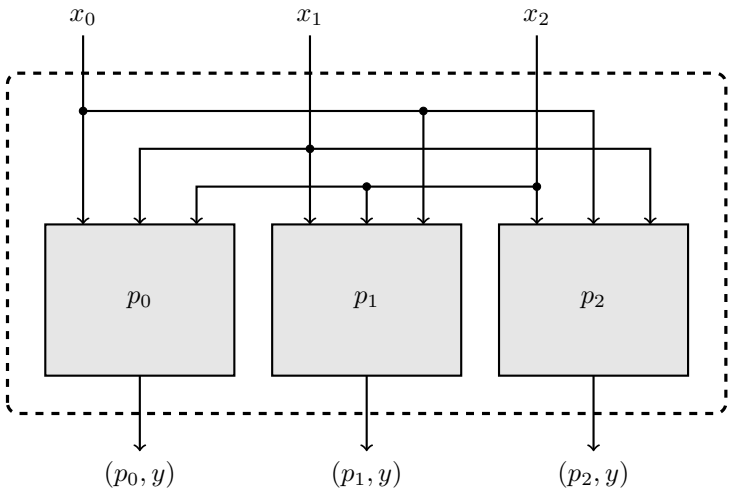

Figure 1 A simple rotation-symmetric architecture.

in a particular architecture. Distributed synthesis is well-studied [25, 21, 14, 15, 16, 9, 24]. However, the approach presented in this paper is the first to synthesize *symmetric* implementations. We consider *rotation-symmetric* system architectures. Rotation-symmetric architectures are multi-process architectures where all processes have access to all system inputs, but see different rotations of the inputs. Figure 1 shows a simple rotation-symmetric architecture. Rotation-symmetric architectures are suitable to reason about distributed systems that lack a central coordination process. They can, for example, model leader election scenarios and distributed traffic light controllers [6]. The fact that the processes obtain their input in different rotations is important: since all processes have the same implementation, they would otherwise also produce the same output. The synthesis problem for such systems could trivially be reduced to the standard synthesis problem by adding a constraint that the outputs are the same all the time.

We present an algorithm for the synthesis of symmetric systems in rotation-symmetric architectures from specifications in linear-time temporal logic (LTL). Most standard synthesis algorithms follow the automata-theoretic approach [22], whereby the given temporal formula is translated into a tree automaton that accepts exactly those computation trees that satisfy the formula. Hence, the specification is realizable if and only if the language of the automaton is non-empty. The synthesis algorithm then simply extracts some finite-state implementation from the language of the automaton. The situation is more difficult when we wish to decide the existence of a symmetric solution, because the language of the automaton may contain both computation trees that belong to symmetric implementations and computation trees that belong to asymmetric implementations. As we show in Section 4, symmetry is *not* a regular property: we therefore cannot check symmetry with a separate tree automaton or encode symmetry as a temporal logic formula and add it to the specification.

The key insight of our algorithm is that the paths in the computation trees produced by symmetric implementations are guaranteed to be invariant under rotations: if, in each position of two (finite or infinite) computation paths, the values of the input variables of the jth process in the first path correspond to the values of the input variables of the $((j + k) \bmod n)$th process, for some k, in the second path, then the values of the output variables of the jth process must also, in each position, correspond to the values of the output variables of the $((j + k) \bmod n)$th process (for all $0 \leq j < n$, where n is the number of

processes). Our algorithm exploits this observation to simplify the computation trees. Paths that are just rotations of each other are collapsed into a single representative. Computations in different processes that must lead to identical outputs are thus kept in the same path of the reduced tree; the paths only split when the symmetry is broken by some input. While symmetry is difficult to check on the original computation tree, it becomes a local condition on individual paths in the reduced tree: as long as the output never spontaneously introduces asymmetry, i.e., as long as every asymmetry in the output can be explained by a previous asymmetry in the input, the reduced tree can be expanded into a full computation tree that we know, by construction, to be symmetric.

As we show in Section 4, the running time of our synthesis algorithm is single-exponential in the number of processes. In Section 5, we show that our algorithm is asymptotically optimal: the problem is EXPTIME-complete in the number of processes. In Section 6, we study the extension of the synthesis problem to the case where the processes no longer have access to all variables. Here, our result is negative: under incomplete information, the symmetric synthesis problem is undecidable even for system architectures where the standard synthesis problem is decidable. This paper is based on previously unpublished results from the first author's PhD thesis [6], where also additional details of the presented results can be found. A full version of this paper with additional proofs is also available [7].

2 Preliminaries

A *reactive system* produces a valuation to the output propositions in some set AP^O and reads the values of the input propositions in some set AP^I in every step of its execution. The behavior of a reactive system can be described as a *computation tree* $\langle T, \tau \rangle$, where $T = (2^{\mathsf{AP}^I})^*$ is the set of tree nodes and $\tau : T \to 2^{\mathsf{AP}^O}$ labels every tree node t by the output propositions $\tau(t)$ that the system sets to **true** after having read t as its (prefix) input sequence.

A *trace* in a computation tree $\langle T, \tau \rangle$ is an infinite sequence $(\tau(\epsilon) \cup t_0)(\tau(t_0) \cup t_1)(\tau(t_0 t_1) \cup t_2)(\tau(t_0 t_1 t_2) \cup t_3) \ldots \in (2^{\mathsf{AP}^I \cup \mathsf{AP}^O})^\omega$. Given some language $L \subseteq (2^{\mathsf{AP}^I \cup \mathsf{AP}^O})^\omega$, *reactive synthesis* is the process of checking if there exists a computation tree $\langle T, \tau \rangle$ with $T = (2^{\mathsf{AP}^I})^*$ as node set such that every trace of $\langle T, \tau \rangle$ is in L. A classical logic to denote specification languages is linear temporal logic (LTL, [19]). LTL formulas for reactive system specifications are built according to the grammar

$$\varphi :== p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\,\varphi \mid \mathsf{G}\,\varphi \mid \mathsf{F}\,\varphi \mid \varphi\,\mathcal{U}\,\varphi,$$

using the temporal operators G (*globally*), F (*eventually*), X (*next*), and $\mathcal{U}$ (*until*). All elements from AP^I and AP^O can be used as propositions p. A more formal definition of LTL is given in [19, 4].

For LTL specifications, it is known that if and only if there exists a computation tree all of whose traces satisfy a specification (i.e., the specification is *realizable*), there exists a *regular* such computation tree. A computation tree is regular if it has only finitely many different sub-trees. Given a computation tree $\langle T, \tau \rangle$, a tree $\langle T', \tau' \rangle$ is a *sub-tree* of $\langle T, \tau \rangle$ if and only if $T = T'$ and there exists a $\hat{t} \in T$ such that for every $t \in T$, we have $\tau'(t) = \tau(\hat{t}t)$. Regular computation trees can be translated to *finite-state machines* and implemented in hardware or software using a finite amount of memory. A tree language for some sets AP^I and AP^O is a subset of all trees $\langle T, \tau \rangle$ with $T = (2^{\mathsf{AP}^I})^*$ and $\tau : T \to 2^{\mathsf{AP}^O}$. A tree or word language is called *regular* if it can be recognized by some finite tree or word automaton (with a *Muller acceptance condition*, see [10] for details).

In *distributed synthesis*, we search for a distributed implementation of a finite state-machine. Given is an architecture that defines several *processes* and the *signals* that connect the processes among themselves and with the global input and output of the architecture. Starting from a specification over all signals, we search for implementations for all of the processes such that the computation tree *induced* by the process implementations and the architecture satisfies the specification. In the induced computation tree, all processes are executed at the same time and in parallel, using the usual parallel composition semantics.

It is known since the seminal work by Pneuli and Rosner [21] that not all architectures have a decidable distributed synthesis problem. Figure 2 depicts the *A0 architecture* that they defined as an example for an undecidable architecture. Finkbeiner and Schewe [9] later proved that the distributed synthesis problem is decidable if and only if there exists no *information fork* in the architecture. An information fork is a pair of processes that are incomparably informed, i.e., for which each of the processes has access to some global input that the other process cannot read. For a more formal definition of distributed synthesis, the interested reader is referred to [9].

A *Turing machine* is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, g)$ in which Q is a finite set of states, Σ is an input alphabet, $\Gamma \supseteq \Sigma$ is a (finite) tape alphabet, $\delta : Q \times \Gamma \to (Q \times \Gamma \times \{-1, 0, 1\})^2$ encodes the Turing machine transition function, $q_0 \in Q$ is an initial state, and g maps every state to its type, which can be *accepting*, *rejecting*, or *transient*. The δ function maps every state/tape content combination to exactly two possible successor state/tape content/tape motion combinations. For deterministic Turing machines, the two successor combinations are always the same. *Alternating Turing machines* [3] extend the non-deterministic Turing machines by partitioning the transient states into *universally branching* and *existentially branching states*. An (alternating) Turing machine accepts a word $w \in \Sigma^*$ if there exists an accepting run tree when starting in state q_0 with the tape empty except for a copy of w where the machine head starts on the first character of w. In all universal states, the Turing machine execution must be accepting for both possible transitions.

We assume that the *modulo function* always returns a non-negative number, such that, e.g., $-13 \mod 5 = 2$.

3 The Symmetric Synthesis Problem

We consider distributed reactive synthesis problems in which all processes share the same implementation. A process has an *interface* $\mathcal{N} = (\mathsf{AP}^I, \mathsf{AP}^O)$ with the local input proposition set AP^I and a local output proposition set AP^O. The connections between the processes are described in an *architecture*.

▶ **Definition 1** (Symmetric architecture). Given an interface $\mathcal{N} = (\mathsf{AP}^I, \mathsf{AP}^O)$, a *symmetric architecture* over $\mathcal{N}$ is a tuple $\mathcal{E} = (S, P, \mathsf{AP}^I_G, E^{in}, E^{out})$ with:

- the set of (internal) signals S,
- the *process set* P,
- the *global input signal set* AP^I_G,
- the *input edge function* $E^{in} : (P \times \mathsf{AP}^I) \to (S \cup \mathsf{AP}^I_G)$, and
- the *output edge function* $E^{out} : (P \times \mathsf{AP}^O) \to S$.

As an example, the architecture given in the right part of Figure 2 hosts processes with the interface $\mathcal{N} = (\{a\}, \{b\})$ and has the components $S = \{y, z\}$, $P = \{0, 1\}$, $\mathsf{AP}^I_G = \{x\}$, $E^{in} = \{(0, a) \mapsto x, (1, a) \mapsto y\}$, and $E^{out} = \{(0, b) \mapsto y, (1, b) \mapsto z\}$. We only consider architectures in which every internal signal is written to by exactly one local output of one process. Given a FSM for a process with an interface $\mathcal{N}$ and an architecture $\mathcal{E} = (S, P, \mathsf{AP}^I_G, E^{in}, E^{out})$ over

$\mathcal{N}$, we can construct an FSM with AP_G^I as input proposition set and S as output proposition set that implements the behavior of the complete architecture when using the FSM as process implementation. Without loss of generality, we use the standard synchronous composition semantics to do so. We define the symmetric synthesis problem as follows:

▶ **Definition 2.** Given an interface $\mathcal{N} = (\mathsf{AP}^I, \mathsf{AP}^O)$, an architecture $\mathcal{E} = (S, P, \mathsf{AP}_G^I, E^{in}, E^{out})$, and a specification φ over the propositions $\mathsf{AP}_G^I \cup S$, *the symmetric synthesis problem* is to check if an FSM implementation $\mathcal{F}$ with the input proposition set AP^I and output proposition set AP^O exists such that the FSM obtained by plugging $\mathcal{F}$ into $\mathcal{E}$ satisfies φ. In case of a positive answer, we also want to obtain $\mathcal{F}$.

4 Rotation-Symmetric Synthesis

Many symmetric architectures found in practice consist of a ring of processes, all of which read all the input to the overall system. A slight generalization of this architecture shape is the class of *rotation-symmetric architectures*.

▶ **Definition 3.** A symmetric architecture $\mathcal{E} = (S, P, \mathsf{AP}_G^I, E^{in}, E^{out})$ over the interface $\mathcal{N} = (\mathsf{AP}^I, \mathsf{AP}^O)$ with n processes is called *rotation-symmetric* if and only if there exists a *local designated proposition set* AP_L^I for every process instance such that the following conditions hold:

- $\mathsf{AP}_G^I = \mathsf{AP}^I = \mathsf{AP}_L^I \times \{0, \dots, n-1\}$ and $P = \{p_0, \dots, p_{n-1}\}$.
- $S = \mathsf{AP}^O \times \{0, \dots, n-1\}$
- for every $p_i \in P$, every $x \in \mathsf{AP}_L^I$, and every $j \in \{0, \dots, n-1\}$, we have $E^{in}(p_i, (x, j)) = (x, (j - i) \mod n)$, and
- for every $x \in \mathsf{AP}^O$ and $p_i \in P$, we have $E^{out}(p_i, x) = (x, i)$.

We show in this section that the symmetric synthesis problem for rotation-symmetric architectures and linear-time temporal logic (LTL) is decidable.

The key observation that we use to prove decidability is that the computation trees that characterize the input/output behavior of a process implementation plugged into a rotation-symmetric architecture have a useful property that we call the *symmetry* property. While this property is non-regular and thus cannot be encoded into the specification (Lemma 6), we show how to decompose it into two sub-properties, one of which is regular. The other one is still non-regular, but has the advantage that we can enforce it in a synthesis process by post-processing the computation tree obtained from a synthesis procedure to contain only rotations of the computation tree paths along so-called *normalized inputs*. Since every tree with the symmetry property is left unaltered by this step and we also describe how to ensure that the result of the post-processing step is guaranteed to be a correct solution, this approach is sound and complete.

We assume some fixed rotation-symmetric architecture $\mathcal{E} = (S, P, \mathsf{AP}_G^I, E^{in}, E^{out})$ over some local process interface $(\mathsf{AP}_L^I, \mathsf{AP}^O)$ to be given, define $\mathcal{I} = 2^{AP_G^I}$ to denote the global input alphabet to all processes, while $\mathcal{O} = 2^{\{\mathsf{AP}^O \times \{0,\dots,n-1\}\}}$ denotes the global output. The local output of one process is given as $O = 2^{\mathsf{AP}^O}$.

The following rotation function will become useful in the analysis below. Let $U = 2^{\mathsf{AP} \times \{0,\dots,n-1\}}$ for some other set AP. We define a *rotation operator* $\mathrm{rot} : U \times \mathbb{Z} \to U$ with $\mathrm{rot}(u, k) = \{(p, (j+k) \bmod n) \mid (p, j) \in u\}$ for every $u \in U$ and $k \in \mathbb{Z}$. Furthermore, we extend the rot function to LTL formulas and define $\mathrm{rot}(\psi, k)$ for an LTL formula ψ over the set of propositions $\mathsf{AP} \times \{0, \dots, n-1\}$ and $k \in \mathbb{Z}$ to be ψ with all atomic propositions (p, j) replaced by $(p, (j+k) \bmod n)$ for $p \in \mathsf{AP}$, $j \in \mathbb{Z}$. For clarity, when dealing with the rot function for some

set $U = 2^{\mathsf{AP}\times\{0,\dots,n-1\}}$, we often partition the elements of $\mathsf{AP}\times\{0,\dots,n-1\}$ by their process indices and for example write $(X_0,\dots,X_{n-1})$ instead of $(X_0\times\{0\})\cup\ldots\cup(X_{n-1}\times\{n-1\})$ for $X_0,\dots,X_{n-1}\subseteq\mathsf{AP}$. The rotation function is extended to sequences of elements in U by rotating the individual sequence items.

▶ **Definition 4** (Symmetry property)**.** Given a tree $\langle T,\tau\rangle$ over $T=\mathcal{I}^*$ and $\tau: T\to\mathcal{O}$, we say that the tree has the *symmetry property* if for each $t\in T$ and $0\le i<n$, $\tau(\mathrm{rot}(t,i)) = \mathrm{rot}(\tau(t),i)$.

▶ **Lemma 5** (Symmetry lemma)**.** *The set of regular trees having the symmetry property is precisely the same as the set of trees that are induced by a rotation-symmetric architecture for some process implementation.*

A proof of the lemma can be found in the full version of this paper [7]. The symmetry property is not a regular tree property, and hence cannot be encoded into a tree or word automaton.

▶ **Lemma 6.** *The set of symmetric computation trees for the two-process rotation-symmetric architecture with process interface* $\mathcal{N} = (\mathsf{AP}_L^I\times\{0,1\},\mathsf{AP}^O)$ *and* $\mathsf{AP}_L^I=\{i\}$ *and* $\mathsf{AP}^O=\{o\}$ *is not a regular tree language.*

Proof. For a proof by contradiction, suppose that the set of symmetric computation trees is regular. The language includes a tree with the symmetry property in which the node labels on the path $(\emptyset,\{i\})^*$ and, symmetrically, on the path $(\{i\},\emptyset)^*$ form the sequence $l=(\emptyset,\emptyset)^1(\{o\},\{o\})(\emptyset,\emptyset)^2(\{o\},\{o\})\ldots$, i.e., the length of the $(\emptyset,\emptyset)$-sequences grows according to the distance to the root. According to the pumping lemma for regular tree languages, however, the sequence l can be partitioned into $l=u\cdot v\cdot w$, such that, for every $k>0$, there exists a tree in the language where the label sequence on $(\emptyset,\{i\})^*$ is $l=u\cdot v^k\cdot w$, while the label sequence on $(\{i\},\emptyset)^*$ is still l. Clearly, these trees are not symmetric. ◀

Since the symmetry property is non-regular, we need to alter the synthesis process itself to account for it. In order to synthesize an implementation for *one* process, we synthesize implementations for *all* processes together. These only need to work correctly on *normalized input sequences* $t\in\mathcal{I}^*$. An input sequence is *normalized* if $\min_i \mathrm{rot}(t,i)=t$, where the min function uses the lexicographic ordering over the strings in $\mathcal{I}^*$. For the ordering of the elements in $\mathcal{I}$, we consider the lexicographic ordering of their tuple representation. For example, we have $(0,1,0)<(0,1,1)$ and $(0,1,0)<(1,0,0)$ for a three-process architecture. A tree with the symmetry property is fully determined by the labels along normalized input sequences, as for every non-normalized input sequence $t'\in\mathcal{I}^*$, we have $\tau(t')=\mathrm{rot}(\tau(t),i)$ for every i such that $t'=\mathrm{rot}(t,i)$.

When only considering the normalized input sequences during synthesis, we can take the computation tree for all processes in the architecture together and complete it by filling all other tree labels with rotations of the tree labels along normalized inputs. We call the resulting tree its *symmetric completion*. If afterwards, we have $\tau(\mathrm{rot}(t,i))=\mathrm{rot}(\tau(t),i)$ for all $t\in\mathcal{I}^*$ and $i\in\mathbb{N}$, then the symmetry lemma guarantees that the resulting tree is induced by some process instantiated in a rotation-symmetric architecture. So if we can guarantee that (1) $\tau(\mathrm{rot}(t,i))=\mathrm{rot}(\tau(t),i)$ is actually the case for all normalized t and $i\in\mathbb{N}$ and (2) that the symmetric completion of the tree satisfies the specification along all paths, then we can obtain a correct process implementation by synthesizing a computation tree for the complete architecture. Our construction for symmetric synthesis consist of these two components, which we describe in more detail below.

4.1 Ensuring Symmetric Completability

Not every $\mathcal{O}$-labeled computation tree can easily be made symmetric by replacing the tree labels for non-normalized input sequences. Take for example a tree $\langle T, \tau \rangle$ for the architecture given in Figure 1 with $\tau(\epsilon) = (\emptyset, \emptyset, \{y\})$. Since the output of the processes is initially different, this means that they cannot have the same implementation. We show in this section that detecting such cases is simple, and the formalization of the observation is a regular property that can be easily encoded into LTL.

▶ **Definition 7.** Let AP be some set, and $P = \{p_0, \dots, p_{n-1}\}$ be a list of process identifiers. For every $x \subseteq (\mathsf{AP} \times \{0, \dots, n-1\})$ and $w = w_0 w_1 w_2 \dots w_l \in (2^{\mathsf{AP} \times \{0,\dots,n-1\}})^*$, we define

$$\begin{aligned} \mathrm{rep}(x) &= |\{j \in \{0, \dots, n-1\} \mid \mathrm{rot}(x, j) = x\}| \\ \mathrm{rep}_S(\epsilon) &= n \\ \mathrm{rep}_S(w) &= \gcd(\mathrm{rep}_S(w_0 \dots w_{n-1}), \mathrm{rep}(w_n)), \end{aligned}$$

where gcd denotes the *greatest common divisor* function.

For some word $t \in \mathcal{I}^*$, $\mathrm{rep}_S(t)$ represents how many different rotations in $\{0, \dots, n-1\}$ of t exist that map the word to itself.

▶ **Lemma 8** (Second symmetry lemma). *Let $\langle T, \tau \rangle$ be a computation tree with $T = \mathcal{I}^*$ and $\tau : T \to \mathcal{O}$ for which for every $t \in T$, we have that $\mathrm{rep}_S(t) \mid \mathrm{rep}_S(\tau(t))$ (where the $\mid$ symbol refers to division without remainder). The unique symmetric completion of $\langle T, \tau \rangle$ has the symmetry property. Furthermore, if $\langle T, \tau \rangle$ is regular, then so is its unique symmetric completion.*

By the second symmetry lemma, it suffices for a computation tree to have $\mathrm{rep}_S(t) \mid \mathrm{rep}_S(\tau(t))$ for all $t \in T$ to ensure that the symmetric completion of the tree has the symmetry property. We can encode this requirement in LTL as

$$\varphi_{outcond} = \bigwedge_{d \in \{1,\dots,n\}, d \mid n} \neg(\mathrm{sym}(\mathcal{I}, d, n) \,\mathcal{U}\, \neg \mathrm{sym}(\mathcal{O}, d, n))$$

for the function

$$\mathrm{sym}(\mathsf{AP}, d, n) = \bigwedge_{a \in \mathsf{AP}, j \in \{0,\dots,n-1\}} (a, j) \leftrightarrow (a, j + \frac{n}{d})$$

that encodes, for each $i \subseteq \mathsf{AP} \times \{0, \dots, n-1\}$ whether $d \mid \mathrm{rep}(i)$ (for $d \in \mathbb{N}$ with $d \mid n$).

4.2 Ensuring That the Tree Completion Satisfies the Specification

If we have a computation tree $\langle T, \tau \rangle$ all of whose traces satisfy some linear-time specification φ, this does not imply that its rotation-symmetric completion satisfies φ as well. If all traces of $\langle T, \tau \rangle$ however satisfy $\varphi \wedge \mathrm{rot}(\varphi, 1) \wedge \dots, \wedge \mathrm{rot}(\varphi, n-1)$, then since we know that every infinite trace in the rotation-symmetric completion is a rotation of a trace in the original tree by some value $i \in \mathbb{N}$, we know that the rotation-symmetric completion also satisfies φ along every trace. So if we synthesize a tree for $\varphi' = \varphi \wedge \mathrm{rot}(\varphi, 1) \wedge \dots \wedge \mathrm{rot}(\varphi, n-1)$ as specification instead of φ, taking the rotation-symmetric completion maintains φ.

Note that strengthening φ to φ' comes without loss of generality if we are interested in rotation-symmetric implementations. By the symmetry property, if the tree $\langle T, \tau \rangle$ induced

by a rotation-symmetric architecture and a process implementation satisfies φ, then it also satisfies $\mathrm{rot}(\varphi, i)$ for all $i \in \mathbb{N}$ as every rotation of every trace in the tree is also a trace in the tree. Hence, to satisfy φ, it also needs to satisfy $\mathrm{rot}(\varphi, i)$ as otherwise we could take a trace not satisfying $\mathrm{rot}(\varphi, i)$, rotate it by $-i$, and obtain a trace that does not satisfy φ.

4.3 Putting Everything Together

Using the concepts defined above, we are now ready to tie them together to a complete synthesis process. We start with a specification φ over the architecture input propositions AP_G^I and the output proposition set $\mathsf{AP}^O \times \{0, \ldots, n-1\}$ for $|P| = n$.

1. We modify the specification φ to $\varphi' = \varphi \wedge \mathrm{rot}(\varphi, 1) \wedge \ldots \wedge \mathrm{rot}(\varphi, n-1)$.
2. We modify φ' to $\varphi'' = \varphi' \wedge \varphi_{outcond}$ (as described in Section 4.2).
3. We synthesize a regular tree $\langle T, \tau \rangle$ that satisfies φ'' along all paths using a classical reactive synthesis procedure. If there is no such tree, the specification is unrealizable.
4. If a regular computation tree $\langle T, \tau \rangle$ is found, we replace every label along non-normalized directions by rotations of τ's labels along normalized directions to get a tree $\langle T', \tau' \rangle$ with the symmetry property.
5. We cut off the labels of τ' except for the output of the first process in the architecture. The resulting (regular) tree is the synthesized process implementation.

▶ **Proposition 9.** *The above synthesis process from LTL has a complexity that is 2EXPTIME in the length of the specification and exponential-time in the number of processes.*

Proof. We use the automata-theoretic approach to reactive system synthesis from [17, 24] and the concepts defined in these works. We start by translating the specification to a universal co-Büchi word (UCW) automaton, which is of size $2^{O(|\varphi|)}$ in the size of the specification. As UCWs do not blow up under conjunction, executing step 1 from the construction above leads to an automaton of size $n \cdot 2^{O(|\varphi|)}$. A deterministic automaton for the added property in step 2 can be built with at most n states, so executing step 2 leads to at most n additional states, and we obtain an automaton with $n + n \cdot 2^{O(|\varphi|)} = n \cdot 2^{O(|\varphi|)}$ many states. The *bounded synthesis* approach works with specifications given as co-Büchi word automata [24] and takes time exponential in the number of states of the automaton. The overall time complexity so far is thus 2EXPTIME in $|\varphi|$ and exponential in n. Step 4 leads to a blow-up of at most a factor of n^2 and can be done in time polynomial in the number of states in the synthesized finite-state machine (whose size is proportional to the time complexity of the synthesis procedure executed in the previous step). Step 5 is simple and takes time linear in the size of the FSM. ◀

Note that even though the construction above discards all non-normalized parts of the synthesized computation tree, asking the synthesis algorithm to nevertheless synthesize these parts according to the specification comes without loss of generality, as trees with the symmetry property (which we are actually searching for) fulfill φ'' along all paths if all of their paths satisfy φ. So the synthesis process does not report spurious unrealizability.

5 Rotation-Symmetric Synthesis – Complexity

The symmetric synthesis construction from the previous section has a time complexity that is doubly-exponential in the length of the specification and singly-exponential in the number of processes. We want to show in this section that this matches the complexity of the problem by giving a corresponding hardness result. The 2EXPTIME-hardness in the specification

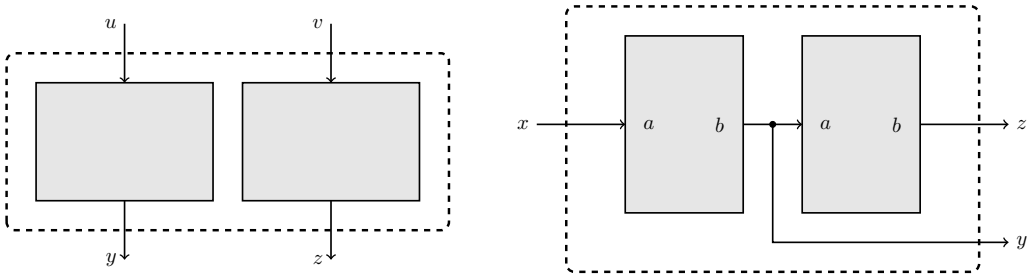

Figure 2 System architectures with undecidable synthesis problems. On the left: architecture A0, as defined by Pnueli and Rosner [21]; on the right: the symmetric architecture S0. The distributed synthesis problem of A0 and the symmetric synthesis problem of S0 are undecidable.

length is inherited from the complexity of LTL synthesis [20]. For the EXPTIME complexity in the number of processes, we provide the following result:

▶ **Lemma 10.** *Given an $f(k)$-space bounded alternating Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, g)$, we can reduce the acceptance of a word $w \in \Sigma^k$ by M to the symmetric realizability problem of $n = f(k)$ processes with a specification in LTL of size polynomial in $|Q| \cdot |\Gamma| \cdot |w|$.*

Proof. We build a specification that requires the processes to output the Turing tape configuration along an execution of the machine. The specification is realizable if and only if the Turing machine does **not** accept the word. Every process outputs the value of one Turing tape cell and if the tape head is at the cell, also the state of the Turing machine. There are n input signals to the architecture, and when the processes start, the left-most local input signals of the processes is used to tell one or more processes that the Turing tape computation should start at that cell with the tape head being initially there (with w as the initial tape content). To account for the rotation-symmetry, the processes output not only the tape content and tape head position, but also the current boundaries of the tape. The specification is modeled such that if start and end markers collide, the simulation of the Turing machine can stop.

The specification also includes conjuncts that require all processes together to simulate the Turing machine computation correctly and to never reach an accepting state. Whenever the alternating Turing machine branches universally, the left-most local process input signal is used to select which successor state is picked. In case of existential branching, the processes can decide which successor state to pick. Enforcing the specification to be realizable if and only if the word w is **not** accepted by the Turing machine helps with taking care of the diverging computations of the Turing machine and those computations that exceed the space bound. Both count as non-accepting in the definition of space-bounded Turing machines. Since these runs never visit accepting states and/or permit the simulation to stop, they are allowed to be simulated by a synthesized implementation.

The specification can be written with size polynomial in $|Q| \cdot |\Gamma| \cdot |w|$ as we only need to define the specification for one process. By the symmetry of the architecture, the other processes have to fulfill it as well. ◀

A more detailed proof can be found in the full version of this paper [7].

▶ **Corollary 11.** *The rotation-symmetric realizability problem (for LTL) has a time complexity that is exponential in the number of processes.*

Proof. Given the question whether a word $w = w_0 \dots w_{k-1}$ is in the language defined by some $(c+1)$-EXPTIME = (c)-AEXPSPACE problem for some $c \in \mathbb{N}$, we can reduce it to the

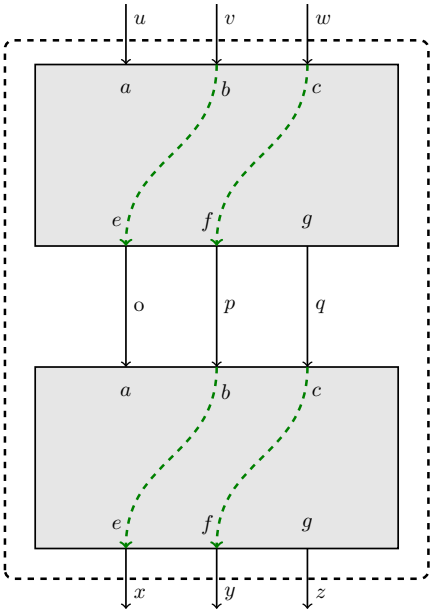

Figure 3 Symmetric architecture S2. The symmetric synthesis problem for S2 is undecidable. The dashed arrows in the process boxes show how the specification given in the proof of Lemma 13 requires the processes to forward the local input streams.

symmetric realizability problem for an LTL specification of length polynomial in k and with a number of processes that is (c)-exponential in k. Since by the space hierarchy theorem [23], the (c)-EXPTIME hierarchy is strict for increasing c, we can conclude that in general, we cannot solve the symmetric realizability problem faster than in time exponential in the number of components. ◀

6 The General Case – Undecidability

The synthesis problem for standard, not necessarily symmetric, distributed systems is decidable as long as the processes can be ordered with respect to their relative knowledge about the system inputs [9]. The problem becomes undecidable as soon as it contains an information fork, i.e., a pair of processes with incomparable knowledge. The simplest such architecture is Pnueli and Rosner's A0 architecture [21], shown on the left in Fig. 2. In this section, we show that for symmetric synthesis, even architectures without information forks, such as the S0 architecture shown on the right in Fig. 2, are undecidable. Our proof is based on Pnueli and Rosner's undecidability argument for A0:

▶ **Lemma 12** ([21]). *For a given Turing machine M, there exists an LTL formula ψ that is realizable in the distributed architecture A0 if and only if M halts and such that the two processes of the unique implementation of M sequentially output binary encodings of the configurations of the Turing machine on y (or z, respectively) upon the first* **true** *value on the input u (or v, respectively).*

Because of the undecidability of the halting problem, Lemma 12 means that the distributed synthesis problem of architecture A0 is undecidable. We prove the undecidability of the symmetric synthesis problem of architecture S0 in two steps. First, we establish the undecidability of the larger architecture S2, depicted in Figure 3, by showing that the

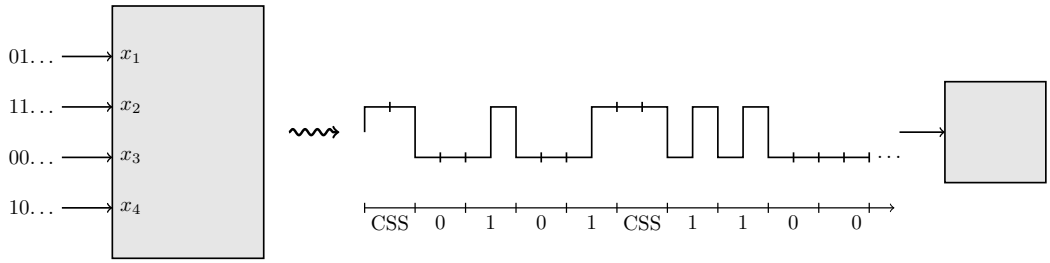

Figure 4 An example for compressing a word with $|\mathsf{AP}| = 4$.

realizability of ψ in A0 can be reduced to the symmetric realizability of an LTL formula over S2; in the second step, we encode the synthesis problem of S0 into the synthesis problem of S2 and thus establish that the synthesis problem for the simpler architecture S0 is undecidable as well.

▶ **Lemma 13.** *The symmetric synthesis problem for architecture S2 is undecidable.*

Proof. We show that there exists an implementation for the specification ψ in the A0 architecture if and only if there exists a joint implementation for the two processes in the S2 architecture that satisfies $\psi' = \psi_d \wedge \mathsf{G}(v \leftrightarrow \mathsf{X}o) \wedge \mathsf{G}(w \leftrightarrow \mathsf{X}p)$, where ψ_d results from prefixing all occurrences of the signals y and z in ψ with a next-time operator.

The results of the two synthesis problems can be translated into each other. A distributed implementation of ψ over A0 is necessarily symmetric: both processes output the same bitstream when reading a **true** value as their local input for the first time. To obtain an implementation for S2, we simulate the process with input a and use g as the local output. Additionally, we copy all values from b to e, and c to f.

Conversely, an implementation found by the symmetric synthesis of S2 provides an implementation of ψ in A0. The key property of the architecture S2 is that the process does not know if the local input b is the (delayed) a input to the other process, or if its c input is the (Turing machine tape) output of the other process. Thus, it cannot find out if it is the top process or the bottom process in the architecture and must prevent violating the specification in either case. A more detailed proof is given in the full version of this paper [7]. ◀

In order to reduce the symmetric synthesis problem of S2 to the symmetric synthesis problem of S0, we introduce compression functions that time-share multiple signals of S2 into a single signal in S0.

Let AP be a set of signals. We call a function $f : (2^{\mathsf{AP}})^\omega \to (2^{\{\chi\}})^\omega$ for some Boolean variable χ a *compression function* if f is injective. We call a function f' that maps a specification over the signal set AP to a different specification over the signal set $\{\chi\}$ the *adjunct compression function* to f if for all $w \in (2^{\mathsf{AP}})^\omega$ and specifications ψ over AP, we have that $w \models \psi$ if and only if $f(w) \models f'(\psi)$.

In the full version of this paper [7], we give such a pair of compression functions for LTL. The compression mechanism is illustrated in Figure 4. One clock cycle in the four-bit-per-character version of a word is spread to 10 computation cycles in the one-bit-per-character version of the word. Every 10 cycles, the 2-cycle *character start sequence* (CSS) $\{\chi\}\{\chi\}$ is instantiated, followed by four two-cycle slots for every signal in AP. Note that the construction ensures that whenever we have $\{\chi\}\{\chi\}\emptyset$ as a part in a compressed word, then we know that a character start sequence begins on the first occurrence of $\{\chi\}$ in this part.

▶ **Theorem 14.** *The symmetric synthesis problem for architecture S0 is undecidable.*

Proof. In order to reduce the symmetric synthesis problem of architecture $S0$ to the symmetric synthesis problem of architecture S2, we compress u, v, w into signal x; o, p, q into signal y; and x, y, z into signal z. A more detailed proof is given in the full version of this paper [7]. ◀

7 Conclusions

In this paper, we have studied the problem of synthesizing symmetric systems. Our new synthesis algorithm is a useful tool in the development of distributed algorithms, because it checks automatically if certain properties in a design problem require symmetry breaking.

Our algorithm synthesizes implementations of rotation-symmetric architectures, i.e., architectures where the processes observe all inputs. The undecidability result for the architecture S0 indicates that it is impossible to extend the synthesis algorithm to architectures where the processes no longer have access to all inputs. A promising direction of research, however, is to use our results to extend existing *semi-algorithms* for synthesis under incomplete information to such symmetric architectures. An example for such an approach is *bounded synthesis* [24], which determines if there exists an implementation with at most n states, where n is a given bound. The specification is translated into a universal co-Büchi automaton, which is then, together with the bound n, encoded into a *satisfiability modulo theory* problem. To ensure correctness under incomplete information, constraints are added that ensure that if a process cannot distinguish two inputs, it transitions to the same successor state. Similarly, for symmetric synthesis, constraints can be added that ensure that the outputs of the individual processes are identical in states that are indistinguishable for them.

Algorithms for symmetric synthesis procedures also offer a new perspective on the problem of synthesizing arbitrarily scalable (i. e. *parametric*) systems. Due to the undecidability of the problem, only very limited solutions to this problem have been found so far. For example, Jacobs and Bloem [11] tackle the case of asynchronous processes with local input in a ring architecture and use the bounded synthesis approach mentioned above. Emerson and Srinivasan [8] present a solution for a multi-process version of a small subset of the temporal logic CTL while Attie and Emerson [2] give a different solution allowing a bigger subset of CTL but only guaranteeing correctness of the solution if certain other conditions are fulfilled, like the dead-lock freeness of the solution produced. In such a setting, symmetric synthesis can be used to detect specifications that are unrealizable even for small system sizes – if there is no solution for a fixed number of processes n, then there is certainly none for scalable systems as well.

References

1 Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Twelfth Annual ACM Symposium on Theory of Computing (STOC)*, pages 82–93, 1980.

2 Paul C. Attie and E. Allen Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, 1998. doi:10.1145/271510.271519.

3 Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

4 E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

5 Shimon Cohen, Daniel J. Lehmann, and Amir Pnueli. Symmetric and economical solutions to the mutual exclusion problem in a distributed system. *Theor. Comput. Sci.*, 34:215–225, 1984.

6 Rüdiger Ehlers. *Symmetric and efficient synthesis*. PhD thesis, Saarland University, 2013. URL: `http://scidok.sulb.uni-saarland.de/volltexte/2013/5607/`.

7 Rüdiger Ehlers and Bernd Finkbeiner. Symmetric synthesis. *ArXiV/CoRR*, 1710.05633, 2017. Full version of this paper.

8 E. Allen Emerson and Jai Srinivasan. A decidable temporal logic to reason about many processes. In *Proc. PODC*, pages 233–246, 1990.

9 Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *Proc. LICS*, pages 321–330, 2005.

10 Jörg Flum, Erich Grädel, and Thomas Wilke, editors. *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*, volume 2 of *Texts in Logic and Games*. Amsterdam University Press, 2008.

11 Swen Jacobs and Roderick Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10(1), 2014. `doi:10.2168/LMCS-10(1:12)2014`.

12 Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *Proc. PODC*, pages 13–22. ACM, 1985.

13 Evangelos Kranakis. Invited talk: Symmetry and computability in anonymous networks. In Nicola Santoro and Paul G. Spirakis, editors, *Proc. SIROCCO*, pages 1–16. Carleton Scientific, 1996.

14 Orna Kupferman and Moshe Y. Vardi. Synthesis with incomplete information. In *Proc. ICTL*, 1997.

15 Orna Kupferman and Moshe Y. Vardi. μ-calculus synthesis. In *Proc. MFCS*, pages 497–507, 2000.

16 Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001)*, July 2001.

17 Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *FOCS*, pages 531–542. IEEE, 2005.

18 Daniel J. Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proc. POPL*, 1981.

19 Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

20 Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 652–671. Springer, 1989.

21 Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, volume II, pages 746–757. IEEE, 1990.

22 Michael O. Rabin. *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, 1972.

23 Desh Ranjan, Richard Chang, and Juris Hartmanis. Space bounded computations: Review and new separation results. *Theor. Comput. Sci.*, 80(2):289–302, 1991. `doi:10.1016/0304-3975(91)90391-E`.

24 Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2007.

25 Pierre Wolper. *Synthesis of Communicating Processes from Temporal-Logic Specifications*. PhD thesis, Stanford University, 1982.

26 Masafumi Yamashita and Tiko Kameda. Computing on an anonymous network. In *Proc. PODC*, pages 117–130, 1988.