Synthesis of Fault-Tolerant Distributed Systems*

Rayna Dimitrova** and Bernd Finkbeiner

Saarland University, Germany

Abstract. A distributed system is fault-tolerant if it continues to perform correctly even when a subset of the processes becomes faulty. Fault-tolerance is highly desirable but often difficult to implement. In this paper, we investigate fault-tolerant *synthesis*, i.e., the problem of determining whether a given temporal specification can be implemented as a fault-tolerant distributed system. As in standard distributed synthesis, we assume that the specification of the correct behaviors is given as a temporal formula over the externally visible variables. Additionally, we introduce the *fault-tolerance specification*, a CTL* formula describing the effects and the duration of faults. If, at some point in time, a process becomes faulty, it becomes part of the external environment and its further behavior is only restricted by the fault-tolerance specification. This allows us to model a large variety of fault types. Our method accounts for the effect of faults on the values communicated by the processes, and, hence, on the information available to the non-faulty processes. We prove that for fully connected system architectures, i.e., for systems where each pair of processes is connected by a communication link, the fault-tolerant synthesis problem from CTL* specifications is 2EXPTIME-complete.

1 Introduction

Fault-tolerance is an important design consideration in distributed systems. A fault-tolerant system is able to withstand situations where a subset of its components breaks: depending on the chosen type of fault-tolerance, the system may completely mask the fault, return to correct behavior after a finite amount of time, or switch to a behavior that is still safe but possibly less performant. Fault-tolerance is highly desirable but often difficult to implement. Thus, formal methods for verification [7] and synthesis [10] of fault-tolerance are necessary.

Traditionally, fault-tolerance requirements are chosen manually. While it is obviously desirable to stay as close as possible to the normal behavior, the question which type of fault-tolerance can be realized in a given system is difficult to decide and requires a careful analysis of both the desired system functionality and the possible faults. In this paper, we develop algorithmic support for this

---

* This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center Automatic Verification and Analysis of Complex Systems (SFB/TR 14 AVACS).

** Supported by a Microsoft Research European PhD Scholarship.

design step. We present a synthesis algorithm that determines if a given temporal specification has a fault-tolerant implementation, and, in case the answer is positive, automatically derives such an implementation.

Our goal is thus more ambitious than previous approaches (cf. [10, 2, 4]) to fault-tolerant synthesis, which transform an existing fault-intolerant implementation into a fault-tolerant version. While such approaches are often able to deliver fault-tolerant systems, they are inherently incomplete, and can therefore not be used to *decide* whether a given fault-tolerance requirement can be realized. In order to obtain a decision procedure, we cannot treat the implementation of the system functionality and the implementation of its fault-tolerance as two separate tasks, but must rather extend the synthesis algorithm to address both concerns at once. In the restricted setting of closed systems, i.e., of systems without input, such a combination has already been carried out: Attie et al. [1] represent faults by a finite set of fault actions that may be carried out by a malicious environment. Their method then synthesizes a program that is correct with respect to a specified set of such possible environments.

The key challenge in moving from simple closed systems to general distributed systems is to account for the incomplete information available to the individual processes. Faults may affect the communication between processes, which affects the information the non-faulty processes have. Our setting builds on that of standard distributed synthesis [6], where the communication links between the processes are described as a directed graph, called the *system architecture*. We assume the architecture is *fully connected* and the system specification is *external* [13, 8], i.e., it does not refer to the internal variables. For standard synthesis, this case is known to be decidable: while the processes may read different inputs, they can simply transmit all information to the other processes through the internal communication links. The distributed system thus resembles a monolithic program in the sense that all processes are aware of the global state.

The situation is more difficult in a fault-tolerant system, since, when a fault occurs in some process, the process essentially becomes part of the hostile environment and the remaining processes can no longer rely on receiving accurate information about the external input at its site. We present a synthesis algorithm for CTL* specifications that accounts for the resulting incomplete information. The given CTL* specification is a *fault-tolerance specification* which encodes the effects and the durations of the faults and the desired type of tolerance.

Our algorithm is based on a transformation of the architecture and of the fault-tolerance specification. The architecture transformation changes the set of external input variables by introducing a new input variable for each process and making the original input variables unobservable for all processes in the architecture. The transformation of the specification establishes the relation between the original input of a process and the new *faulty input*. The two inputs are constrained to be the same during the normal operation of the corresponding process, which guarantees the correctness of the transformation, and may differ when the process is faulty, which allows us to assume that in the transformed architecture the faults do not affect the transmission of external input. Thus, we

can reduce the distributed synthesis problem for the original architecture and fault-tolerance specification to the one of finding a monolithic implementation that satisfies the transformed specification in the presence of faults.

We hence establish that the synthesis of fault-tolerant distributed systems with fully connected system architectures and external specifications is decidable. In fact, the problem is no more expensive than standard synthesis: fault-tolerant distributed synthesis from CTL* specifications is 2EXPTIME-complete.

2 Modelling Fault-Tolerant Systems

2.1 Faults and Fault-Tolerance

Types of Faults. In the field of fault-tolerant distributed computing faults are categorized in a variety of ways. The categorization of faults according to the behavior they cause, results in several standard classes [3, 1]. *Stuck-at* faults can, for example, cause a component or a wire to be stuck in some state. If a process is affected by a *fail-stop* or a *crash* fault, it stops (potentially permanently) executing any actions before it violates its input-output specification. In both cases the process is uncorrectably corrupted, but while fail-stop faults are *detectable*, that is, other processes are explicitly notified of the fault, crash stops are *undetectable*. If a process fails to respond to an input from another component, i.e., some action is omitted, it is said to exhibit an *omission fault*. Omission faults are a subset of the class of *timing faults* that cause the component to respond with the correct value but outside the required time interval. The most general class of *Byzantine faults* encompasses all possible faults, including arbitrary and even malicious behavior of the affected process, and are in general undetectable.

According to their duration, faults can be *permanent*, *transient*, or *intermittent*. In the latter two cases, upon recovery the affected process returns to normal operation from the arbitrary state it has reached in the presence of the fault.

Fault-Tolerance Requirements. Usually the system is not required to satisfy the original specification after a fault occurs, but instead comply with some fault-tolerance policy. Fault-tolerance properties are generally classified according to whether and how they respect the safety and liveness parts of the original specification. This classification yields three main types of tolerance. *Masking tolerance* always respects both safety and liveness. In *non-masking tolerance*, however, the safety property might be temporarily violated, but is guaranteed to be eventually restored, while the liveness part is again always respected. A third type is *fail-safe tolerance*. When formalizing fail-safe tolerance it is assumed that the original specification is given as conjunction of a safety and a liveness specifications [12], and after fault occurrence only the safety conjunct has to be satisfied.

2.2 Architectures for Fault-Tolerant Synthesis

An architecture describes the communication between the processes in a distributed system and their interaction with the external environment. We model

the occurrence of a fault as an action of the environment. In the following, we assume that faults are *detectable*, that is there exists a reliable unit of the external environment that notifies all processes immediately when a fault occurs in some of them, and also informs them exactly which processes were faulty in the previous execution step. To this end, we consider architectures with a distinguished set of external input *fault-notification variables*, which all processes in the system are allowed to read. Alternatively, the fault-notification variables could be made invisible to the system processes, in which case finding the fault-detection mechanism would be part of the synthesis problem.

An *architecture* $A = (env, P, Ext, C, (D(v))_{v \in Ext}, (In(p), Out(p))_{p \in P})$ is a tuple that consists of: environment env, a finite set of processes P, a set Ext of *external variables* together with a finite domain $D(v)$ for each $v \in Ext$, a set C (disjoint from Ext) of *internal variables*, and read and write permissions for each $p \in P$. The set Ext is the union of the disjoint sets I, O, H and N, where:

– The set I consists of the *external input variables* whose values are supplied by the environment env. The set I is the union of the sets I_p, where for each process p, I_p is a set of external input variables, this process can read. Each variable in I is read by at least one (possibly several) processes in P.

– The set O consists of the *external output variables*, via which the processes provide their output to the environment env. The set O is the union of the *disjoint* sets O_p, where for each process p, O_p is the set of external output variables written by that process, which no other process in P can read.

– The set H consists of the *external private environment variables* written by the environment env, and which none of the processes in P can read.

– The set $N = \{n_p, m_p \mid p \in P\}$ of external input variables for *fault notification* contains one variable n_p for each process p that is used by the environment to notify all processes for a fault occurrence in p and a variable m_p that indicates whether p was faulty in the previous execution step. The variables in N can be read by all processes and are written only by the environment. The domain $D(n_p)$ of n_p is a finite subset of $\mathbb{N}$ that consists of the different faults that can occur in process p, where 0 indicates normal operation. Similarly for m_p.

The set C consists of the variables used for *internal communication* between the processes. It is the union of the *disjoint* sets C_p, where for each process p, C_p is the set of internal variables written by p via which it communicates to the other processes. We denote with V the set of all variables in an architecture A.

For a process p, the set $In(p)$ consists of all variables (internal or external) this process is allowed to read and $Out(p) = C_p \cup O_p$ consists of all variables that this process is allowed to write. By definition, the sets $Out(p)$ are disjoint.

The architecture associates with each external variable $v \in Ext$ a finite nonempty domain $D(v)$ together with some designated element $d_0(v) \in D(v)$. The domains of the internal variables are unconstrained by the architecture, and hence the capacity of the communication channels is not limited a priori.

Consider some nonempty finite domains $D(v)_{v \in C}$ for the internal variables in A. For a subset $U \subseteq V$, we denote with $D(U)$ the Cartesian product $\prod_{u \in U} D(u)$ and with $d_0(U)$ the tuple $(d_0(u))_{u \in U}$. For $d \in D(U)$, $u \in U$ and $U' \subseteq U$, $d\langle u \rangle$ and

$d\langle U' \rangle$ denote the projections of d on the variable u and on the subset of variables U', respectively. For a finite or infinite sequence $\sigma = d_0 d_1 d_2 \dots$ of elements of $D(U)$ and $j \geq 0$, we denote by $\sigma[j]$ the element d_j and with $\sigma(j)$ the prefix of length j of σ (if $j = 0$, then $\sigma(j) = \varepsilon$). Projection trivially extends to sequences and prefixes. When σ is finite, we denote with $|\sigma|$ the number of elements of σ.

We consider synchronous communication with delay: at each step, each process reads its current external input and the output of the processes in P delayed by one step. For a *global computation history* $\sigma \in D(V)^*$ we have that $\sigma[0] = d_0(V)$ and for every $j \geq 1$, $\sigma[j]\langle I \cup H \cup N \rangle$ is the input provided by the environment at step j and $\sigma[j]\langle V \setminus (I \cup H \cup N) \rangle$ is the output of the processes at step $j-1$, i.e., the history reflects the delay. Thus, for simplicity of the presentation we have assumed that the delay of each variable $v \in V \setminus (I \cup H \cup N)$ is 1. Our results can be easily extended to the case of arbitrary *a priori fixed* delays.

Fully Connected Architectures. An architecture A is *fully connected* if every pair of processes is connected via a communication link with sufficient capacity.

From now on, we consider only fully connected architectures and w.l.o.g. assume that $C = \{c_p, t_p \mid p \in P\}$, where for each process p, the variables c_p and t_p are written by p and read by all processes, and the domain of c_p is fixed to be $D(I_p)$. Thus, process p can use c_p to communicate its input. We denote with c_p^v the component of the variable c_p used for the transmission of $v \in I_p$. The domains of the variables t_p for $p \in P$ are left unspecified in the architecture.

2.3 The Specification Language CTL*

Syntax. Let AP be a finite set of atomic propositions. The logic CTL* distinguishes state and path formulas. State formulas are called *CTL* formulas*.

State formulas over AP are formed according to the following grammar, where $p \in AP$ and θ stands for a path formula: $\varphi ::= \mathit{true} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{E}\theta$. Path formulas are formed according to the following grammar, where φ is a state formula and θ, θ_1 and θ_2 are path formulas: $\theta ::= \varphi \mid \neg\theta \mid \theta_1 \wedge \theta_2 \mid \mathsf{X}\theta \mid \theta_1 \,\mathcal{U}\, \theta_2$.

As abbreviations we can define the remaining usual boolean operators over state and path formulas. For a path formula θ, we define the state formula $\mathsf{A}\theta$ as $\neg\mathsf{E}\neg\theta$, the path formula $\mathsf{F}\theta$ as $\mathit{true}\,\mathcal{U}\,\theta$ and the path formula $\mathsf{G}\theta$ as $\neg\mathsf{F}\neg\theta$.

Trees. As usual, for a finite set X, an X-tree is a prefix-closed subset $T \subseteq X^*$ of finite words over X. The direction of every nonempty node $\sigma \cdot x \in X^+$ is defined as $\mathit{dir}(\sigma \cdot x) = x$, and for ε, $\mathit{dir}(\varepsilon) = x_0$ where $x_0 \in X$ is some designated root direction. A X-tree T is called *total* if $\varepsilon \in T$ and for every $\sigma \in T$ there exists at least one successor $\sigma \cdot x \in T$, $x \in X$. If $T = X^*$, then T is called *full*. For a given finite set Y, a Y-labeled X-tree is a pair $\langle T, l \rangle$, where T is an X-tree and $l : T \to Y$ is a labelling function that maps each node in T to an element of Y.

Semantics. Consider a set of variables V with $D(V)$ being the Cartesian product of their domains. Let AP be a finite set of atomic propositions over V. A CTL* formula φ over AP can then be interpreted over total $D(V)$-labeled trees according to the standard semantics [5] of CTL*. A total $D(V)$-labeled tree $\langle T, l \rangle$ is a model of φ, written $\langle T, l \rangle \models \varphi$, iff the root node of $\langle T, l \rangle$ satisfies φ.

2.4 Specifying Fault-Tolerance

The *system specification* describes the desired input-output behavior of the system in the absence of faults and leaves the internal communication unconstrained. That is, we are given an *external specification* as a CTL* formula φ over atomic propositions from the set $AP = \{v = a \mid v \in Ext \setminus N, a \in D(v)\}$, i.e., about external variables. The models of φ are total $D(Ext)$-labeled $D(Ext)$-trees.

In the presence of faults, the system need not satisfy the original specification, but instead comply with some (possibly weaker) *fault-tolerance specification.* An external specification for an architecture A can refer to the fault notification variables in N. This allows for specifying the intended fault-tolerance policy as well as encoding the effects and durations of faults in the input CTL* formula.

Given the original specification φ, we first construct a formula Φ_{TOL} according to the required type of fault-tolerance. The user can describe manually as a CTL* formula the desired properties of the behavior of the system in the presence of particular faults in particular processes and combinations thereof. Of course, classical fault-tolerance requirements, such as masking, non-masking or fail-safe, can be also specified (for masking tolerance it suffices to leave the specification unchanged). Moreover, in the case of simple specifications such as *invariants*, i.e., of the form $\mathsf{AG}\psi$, this compilation can be done *automatically*: For fail-safe and non-masking tolerance, the tolerance properties are respectively $\mathsf{AG}(\psi \vee (\textit{fault-present} \wedge \psi_{safe}))$ and $\mathsf{AG}(\psi \vee (\textit{fault-present} \wedge \mathsf{AFAG}\psi))$, where $\textit{fault-present} = \bigvee_{p \in P} \neg(n_p = 0)$ and ψ_{safe} is the safety conjunct of ψ.

In our model, the occurrence of a fault causes the affected process to behave in an arbitrary way, i.e., it exhibits maximal behavior. However, by constraining this behavior in the fault-tolerance specification, we can model several of the fault types mentioned in the beginning of this section, as well as many more.

Given a set of faults with their effects on the behavior of a process and their durations, we transform the formula φ_{TOL} into the *fault-tolerance specification* Φ^t, by relativizing the path quantifiers in the formula φ_{TOL} w.r.t. the corresponding assumptions on the environment. These assumptions are encoded in the formulas *fault-behavior*, *fault-duration*, and *fault-distribution*, whose construction we discuss below. Thus, the fault-tolerance specification Φ^t is obtained from the formula φ_{TOL} by substituting each occurrence of $\mathsf{A}\theta$ by $\mathsf{A}((\textit{fault-behavior} \wedge \textit{fault-duration} \wedge \textit{fault-distribution}) \rightarrow \theta)$, and each occurrence of $\mathsf{E}\theta$ by $\mathsf{E}(\textit{fault-behavior} \wedge \textit{fault-duration} \wedge \textit{fault-distribution} \wedge \theta)$.

The formula *fault-behavior* describes the possible behaviors of the processes in the presence of each of the given faults. Let $\textit{faulty-output}(d, p)$ be a state formula describing the possible outputs of process p when affected by the fault of type d (we can assume that a stopped process outputs some default element $\perp$). Then, $\textit{fault-behavior} = \mathsf{G} \bigwedge_{p \in P} \bigwedge_{d \in D(n_p)} (n_p = d \rightarrow \mathsf{X}(\textit{faulty-output}(d, p)))$.

The formula *fault-duration* constrains the duration of faults. Let $D'(n_p)$ and $D''(n_p)$ be the subsets of $D(n_p)$ consisting of the permanent and the transient faults, respectively, for a process $p \in P$. For each $d \in D''(n_p)$, we assume the existence of a boolean variable r_p^d in H, where r_p^d being true means that process p has recovered from d. The path formulas $\textit{permanent}(d, p) = \mathsf{G}((n_p = d) \rightarrow \mathsf{G}(n_p =$

d)) and $\mathit{transient}(d,p) = \mathsf{G}(((n_p = d) \rightarrow (n_p = d \,\mathcal{U}\, r_p^d)) \wedge (r_p^d \rightarrow (\neg(n_p = d) \wedge \mathsf{G} r_p^d)))$ state that the occurrence of fault of type d in process p is permanent, respectively transient (i.e., the duration of the fault is finite and the recovered process cannot perturb again, cf. [7]). Finally, we define the formula $\mathit{fault\text{-}duration} = \bigwedge_{p \in P}((\bigwedge_{d \in D'(n_p)} \mathit{permanent}(d,p)) \wedge (\bigwedge_{d \in D''(n_p)} \mathit{transient}(d,p)))$.

The user can also provide a path formula *fault-distribution* that constrains the number of faulty processes in the considered system during the execution, e.g., there is at most one faulty process at every point of the execution.

To avoid restriction to only memoryless implementations, we assume that at every point of the system's execution at least one process is not faulty, i.e., the synthesized system is designed to tolerate up to $n-1$ simultaneously faulty processes, where n is the total number of processes. Thus, we assume that the formula *fault-distribution* is a conjunction of the user specified requirements and: the formula $\mathsf{G} \bigvee_{p \in P}(n_p = 0)$ which guarantees that there is at least one non-faulty process at every point, and the formula $\mathsf{G} \bigwedge_{p \in P}(n_p = d \rightarrow \mathsf{X}(m_p = d))$, which states that the values of the variables m_p and n_p are correctly related.

Note that for common fault types such as fail-stop or stuck-at, as well as for the usual constraints on the duration of faults, the fault-tolerance specification can be compiled automatically from the original specification. The user can also specify customized requirements expressible in CTL*. From now on, we assume that the fault-tolerance specification is given as input to our algorithm.

Example (Reliable Broadcast). In a broadcast protocol, the environment consists of n clients $E_1, \ldots, E_n$, which broadcast messages. The system consists of n servers, $S_1, \ldots, S_n$ that correspond to the processes $p_1, \ldots, p_n$, which deliver the messages to the clients. Each client E_j communicates only with the corresponding server S_j and we assume that each message sent by a client is unique.

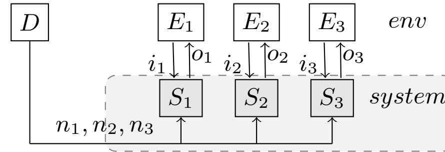

Fig. 1. Architecture of a distributed server.

A system with 3 servers is depicted on Fig. 1, where we have omitted the internal communication variables. Let M be the finite set of possible message contents. The domain of each input, output or communication variable $v \in \{i_j, o_j, c_j \mid j = 1,2,3\}$ is $D(v) = \{(m,j) \mid m \in M, j \in \{1,2,3\}\} \cup \{\bot\}$, where j denotes the broadcaster's name and $\bot$ indicates the absence of message.

In the absence of faults, the correctness can be specified by the following standard requirements: (1) If a client E_j broadcasts a message m, then the server S_j eventually delivers m; (2) If a server delivers a message m, then all

servers eventually deliver m; (3) For every message m, every server delivers (m, l) at most once and does so only if m was previously broadcast by the client E_l.

The environment component D notifies all servers when a fault in server j occurs, by setting n_j to 1. A faulty server sends arbitrary messages to the corresponding client and to the other servers. The duration of faults is unconstrained. Thus, both formulas *fault-behavior* and *fault-duration* are equivalent to *true*.

We specify the fault-tolerance requirement as follows. We replace each requirement that a server eventually delivers a message m by the weaker requirement that it has to eventually deliver a message, provided that from that point on it is never faulty. The safety property that the servers do not invent messages is also weakened to hold only for non-faulty processes. Thus, we obtain a variation of the standard requirements for reliable broadcast from [9].

Validity: *If a client E_j broadcasts a message m and the corresponding server S_j is never faulty from that point on, then S_j eventually delivers m.*

$$\varphi_j^V = \mathsf{AG} \bigwedge_{m \in M} ((i_j = (m, j) \wedge \mathsf{G}(n_j = 0)) \rightarrow \mathsf{F}(o_j = (m, j)))$$

Agreement: *If a non-faulty server E_j delivers a message (m, l), then all servers that are non-faulty from that point on eventually deliver (m, l).*

$$\varphi_j^A = \mathsf{AG} \bigwedge_{m \in M, l \in P} (o_j = (m, l) \wedge n_j = 0 \rightarrow \bigwedge_{p \in P} (\mathsf{G}(n_p = 0) \rightarrow \mathsf{F}(o_p = (m, l))))$$

Integrity: *For every message m, every non-faulty server E_j delivers (m, l) at most once and does so only if m was previously broadcast by the client E_l.*

$$\begin{aligned} \varphi_j^I = {} & \mathsf{AG} \bigwedge_{m \in M, l \in P} ((n_j = 0 \wedge o_j = (m, l)) \rightarrow \mathsf{XG}(n_j = 0 \rightarrow \neg(o_j = (m, l)))) \wedge \\ & \mathsf{AG} \bigwedge_{m \in M, l \in P} \neg((\neg i_l = (m, l)) \,\mathcal{U}\, (o_j = (m, l) \wedge n_j = 0 \wedge \neg i_l = (m, l))) \end{aligned}$$

2.5 The Fault-Tolerant Synthesis Problem

Let $A = (env, P, Ext, C, (D(v))_{v \in Ext}, (In(p), Out(p))_{p \in P})$ be a fully connected architecture. A *distributed implementation* for the architecture A consists of a tuple $(D(v))_{v \in C}$ of sets defining domains for the internal variables and a tuple $\hat{s} = (s_p)_{p \in P}$ of implementations for the processes in P, where an *implementation* for a process p is a function (strategy) $s_p : D(In(p))^* \rightarrow D(Out(p))$ which maps each *local input history* for p to an assignment to the variables written by p. A distributed implementation $\hat{s}$ is *finite-state* if for each process p, the domain $D(C_p)$ is finite and the strategy s_p can be represented by a finite automaton.

In the absence of faults, a distributed implementation $\hat{s} = (s_p)_{p \in P}$ defines a *computation tree* $\mathcal{CT}(\hat{s}) = \langle T, dir \rangle$, where $T \subseteq D(V)^*$ is the greatest total tree such that for all $\sigma \in D(V)^*$ and all $d \in D(V)$, if $\sigma \cdot d \in T$, then $\sigma \in T$ and for every $p \in P$ it holds that $d\langle Out(p) \rangle = s_p(\sigma\langle In(p) \rangle)$. Recall that the *realizability problem* for an architecture A with fixed finite domains for all variables and a CTL* specification φ over $Ext \setminus N$ is to decide whether there exists a finite-state distributed implementation $\hat{s}$ for A such that $\mathcal{CT}(\hat{s}) \models \varphi$. The *distributed synthesis problem* requires finding such an implementation if one exists.

In the presence of faults, a distributed implementation $\hat{s} = (s_p)_{p \in P}$ defines a *fault computation tree* $\mathcal{FCT}(\hat{s}) = \langle T, dir \rangle$, where $T \subseteq D(V)^*$ is the greatest

total tree such that for all $\sigma \in D(V)^*$ and $d \in D(V)$, if $\sigma \cdot d \in T$, then $\sigma \in T$ and *for every* $p \in P$ *such that* $dir(\sigma)\langle n_p \rangle = 0$ it holds that $d\langle Out(p)\rangle = s_p(\sigma\langle In(p)\rangle)$, i.e., the output of only non-faulty processes determines the successors of a node.

The implementations for the processes in P should be independent of information about the external environment the processes do not have. Consider an external input variable $v \in I$ and assume that at some point all processes allowed to read v are faulty. The behavior of the (non-faulty) processes in P at this and at later points of the execution of the system should not depend on the value of v at that moment, because the faulty processes may communicate the value incorrectly and their states may be arbitrarily perturbed. Thus, we say that two local histories for a process p are *equivalent up to faults* if they differ only in the values of external input variables at points when all processes reading them were faulty. A distributed implementation is *consistent w.r.t. faults* if all strategies produce the same output for histories that are equivalent up to faults.

Note that for an implementation that is consistent w.r.t. faults, the output of a strategy for a process p is allowed to depend on the value of a variable $v \in I_p$ from points in the history in which process p was faulty, as long as some process allowed to read v was not faulty. Thus, provided that at each point of the execution of the system there exists at least one process that is not faulty (an assumption that can be specified in the fault-tolerance specification as shown above), a recovered process is allowed to depend on the history up to equivalence w.r.t. faults. This is possible since in a fully connected architecture a process can, upon recovery from a fault, receive information about the current state of the system from another process that was not faulty at the previous step.

Definition 1 (Equivalence up to faults). *For a process* $p \in P$*, we define the equivalence relation* $\equiv^f_p$ *on* $D(In(p))^*$ *in the architecture* A *as follows. For* $\sigma_1 \in D(In(p))^*$ *and* $\sigma_2 \in D(In(p))^*$*, we have* $\sigma_1 \equiv^f_p \sigma_2$ *if and only if* $|\sigma_1| = |\sigma_2|$*,* $\sigma_1[|\sigma_1| - 1] = \sigma_2[|\sigma_2| - 1]$ *and for every* $0 \le j < |\sigma_1| - 1$ *it holds that: (1)* $\sigma_1[j]\langle In(p) \setminus I_p\rangle = \sigma_2[j]\langle In(p) \setminus I_p\rangle$ *and (2) for every* $v \in I_p$*, if there exists a process* $q \in P$ *with* $v \in I_q$ *and* $\sigma[j]\langle n_q\rangle = 0$*, then it holds that* $\sigma_1[j]\langle v\rangle = \sigma_2[j]\langle v\rangle$*.*

Definition 2 (Consistency w.r.t. faults). *We say that a distributed strategy* $\hat{s}$ *for the architecture* A *is* consistent w.r.t. faults *if for every* $p \in P$ *and for every* σ_1 *and* σ_2 *in* $D(In(p))^*$ *with* $\sigma_1 \equiv^f_p \sigma_2$ *it holds that* $s_p(\sigma_1) = s_p(\sigma_2)$*.*

Definition 3 (Fault-Tolerant Synthesis Problem). *The* fault-tolerant realizability problem *for an architecture* A *and a CTL** *fault-tolerance specification* Φ^t *is to decide whether there exists a finite-state distributed implementation* $\hat{s}$ *for* A *that is consistent w.r.t. faults and such that* $\mathcal{FCT}(\hat{s}) \models \Phi^t$*. The* fault-tolerant synthesis problem *requires finding such an implementation if the answer is yes.*

3 Synthesis

Our synthesis algorithm builds on that of single-process synthesis under incomplete information [11], via a standard reduction for fully connected architectures

and external specifications [8]. We now provide the necessary preliminaries based on the classical synthesis case and in the next sections we present the methodology we developed to employ a similar approach in the fault-tolerant setting.

3.1 Synthesis for Fully Connected Architectures

Transmission Delay. In a fully connected architecture, the output of every process p may depend, with a certain delay, on all external input variables in $I \cup N$. The delay, $delay(v, p)$, of the transmission of an external input variable v to a process p in a fully connected architecture is 0 if $v \in I_p \cup N$ and 1 otherwise.

Input-Output Functions. An *input-output function* for a process p is a function $g_p : D(I \cup N)^* \to D(O_p)$ which based on the *global input history* assigns values to the variables in O_p. A *global input-output function* $g : D(I \cup N)^* \to D(O)$ assigns values to all external output variables, based on the global input history.

An input-output function g_p is *delay-compatible* if for each $\sigma_1, \sigma_2 \in D(I \cup N)^*$, for which for every $v \in I \cup N$ it holds that $\sigma_1\langle v\rangle(|\sigma_1| - delay(v, p)) = \sigma_2\langle v\rangle(|\sigma_2| - delay(v, p))$, it holds that $g_p(\sigma_1) = g_p(\sigma_2)$. A global input-output function g is *delay-compatible* iff so is the projection of g on O_p, for every process $p \in P$.

Routing Strategies. Given a nonempty set $D(v)$ for each $v \in C$, a *routing* $\hat{r} = (r_p)_{p \in P}$ for an architecture A is a tuple of local *memoryless* strategies, called *routing strategies*, where for each $p \in P$, $r_p : D(In(p)) \to D(C_p)$ is a function that given values for the variables in $In(p)$, assigns values to the variables in C_p.

In a fully connected architecture A, each process p can transmit the values of I_p to the other processes via the variable c_p. A *simple routing* $\hat{r} = (r_p)_{p \in P}$ is one for which $r_p(d)\langle c_p\rangle = d\langle I_p\rangle$ and it allows every process to trivially reconstruct the value of every variable in I. A distributed implementation $\hat{s}$ for A *has simple routing* if for every $\sigma \in D(V)^*$ and $p \in P$ it holds that $s_p(\sigma\langle In(p)\rangle)\langle c_p\rangle = dir(\sigma)\langle I_p\rangle$, i.e., the strategies directly forward the external input. For fully connected architectures it suffices to consider only implementations with simple routing.

Synthesis for Fully Connected Architectures and External Specifications. In [8] it was shown that the distributed synthesis problem is decidable for uniformly well-connected architectures with linearly preordered information and external specifications, and it can be reduced to finding a collection of delay-compatible input-output functions for the processes in P. Fully connected architectures are a special case of uniformly well-connected architectures in which all processes have the same information and hence fall in this class. Moreover, in order to find such a collection of input-output functions, it suffices to find a delay-compatible global input-output function and use projection to obtain functions for the processes in P. Thus, the problem reduces to the single-process synthesis problem under incomplete information with the additional requirement of delay-compatibility.

3.2 Single-Process Synthesis under Incomplete Information

Let $A = (env, \{p\}, Ext, \emptyset, (D(v))_{v \in Ext}, (I \cup N, O))$ be a single-process architecture and ψ be a CTL* specification over the variables in $V = I \cup H \cup N \cup O$.

Tree Automata. An *alternating parity tree automaton* is a tuple $\mathcal{A} = (Y, X, Q, q_0, \delta, \alpha)$, where Y is a finite alphabet, X is a finite set of directions, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times Y \to \mathbb{B}^+(Q \times X)$ is a transition function that maps a state and an input letter to a positive boolean combination of pairs of states and directions, and a coloring function $\alpha : Q \to \mathit{Col} \subset \mathbb{N}$ that maps each state to some color from a finite set Col. An alternating automaton runs on full Y-labeled X-trees. A *run tree* on a given full Y-labeled X-tree $\langle X^*, l\rangle$ is a $Q \times X^*$-labeled tree where the root is labeled with (q, ε) and where for every node ρ with label (q, σ) the set of children K of ρ satisfies the following properties: (1) for every $\rho' \in K$, the label of ρ' is $(q', \sigma \cdot x)$ for some $q' \in Q$ and $x \in X$ such that (q', x') is an atom of $\delta(q, l(\sigma))$, and (2) the set of atoms defined by the set of children K satisfies $\delta(q, l(\sigma))$. An infinite path fulfills the *parity condition* if the maximal color of the states appearing infinitely often is even. A run tree is accepting if all infinite paths fulfill the parity condition. A full Y-labeled X-tree is *accepted* if it has an accepting run tree. A *nondeterministic automaton* is an alternating automaton, in which, in the DNF of each transition every disjunct contains exactly one (q, x) for every $x \in X$.

Symmetric alternating automata are a variant of alternating automata that run on total Y-labeled X-trees. For a symmetric alternating automaton $\mathcal{S} = (Y, Q, q_0, \delta, \alpha)$, Q,q_0, and α are as above, but the transition function $\delta : Q \times Y \to \mathbb{B}^+(Q \times \{\Box, \Diamond\})$ maps a state and an input letter to a positive boolean combination over atoms that refer to *some*($\Diamond$) or *all*($\Box$) successors in the tree. A *run tree* on a given Y-labeled X-tree $\langle T, l\rangle$ is a $Q \times X^*$-labeled tree where the root is labeled with (q, ε) and where for every node ρ with label (q, σ) the set of children K of ρ satisfies the following properties: (1) for every $\rho' \in K$, the label of ρ' is $(q', \sigma \cdot x)$ for some $q' \in Q$, $x \in X$ and $\sigma \cdot x \in T$ such that $(q', \Box)$ or $(q', \Diamond)$ is an atom of $\delta(q, l(\sigma))$, and (2) interpreting each occurrence of $(q', \Box)$ as $\bigwedge_{x \in X, \sigma \cdot x \in T}(q', x)$ and each occurrence of $(q', \Diamond)$ as $\bigvee_{x \in X, \sigma \cdot x \in T}(q', x)$, the set of atoms defined by the set of children K satisfies $\delta(q, l(\sigma))$.

Automata-Theoretic Solution. For a CTL* formula ψ one can construct an alternating parity tree automaton $\mathcal{A}_\psi$ with $2^{O(|\psi|)}$ states that accepts exactly the models of ψ [11]. Via the automata transformations described in [11], the realizability problem for the single process architecture A and the specification ψ can be reduced to the nonemptiness of an alternating tree automaton $\mathcal{C}_\psi$ that is obtained from $\mathcal{A}_\psi$ and that has the same number of states as $\mathcal{A}_\psi$.

Theorem 1 (from [11]). *The single-process synthesis problem for CTL* with incomplete information is 2EXPTIME-complete.*

4 Encoding Fault-Tolerant Realizability

In this section we present a transformation of the architecture and the fault-tolerance specification to an architecture and a specification for which the distributed fault-tolerant synthesis problem can be reduced to single-process synthesis under incomplete information. The key challenge is to account for the

fact, that the non-faulty processes cannot rely on receiving accurate information about the external input of a faulty process. We describe how we model the effect of a fault occurrence on the informedness of the non-faulty processes. We show that the described transformations do not affect fault-tolerant realizability.

Architecture Transformation. To circumvent the change of informedness of the processes in the architecture caused by the occurrences of faults, and yet allow a faulty process to communicate incorrectly its external input to the other processes, we introduce a *faulty copy of the external input of each process*.

Formally, we transform the architecture A into an architecture A^f defined as follows: $A^f = (env, P, Ext^f, C, (D(v))_{v \in Ext^f}, (In^f(p), Out(p))_{p \in P})$, where V^f is partitioned into $I^f = F$, $H^f = H \cup I$, N, C and O. The new set of variables $F = \{f_p \mid p \in P\}$ consists of the *external faulty-input variables*, one for each process p in P, whose values are supplied by the environment. For $p \in P$ we have $In^f(p) = (In(p) \setminus I_p) \cup \{f_p\}$. The domain of each f_p is $D(I_p)$ and we denote with f_p^v the component of f_p that corresponds to a variable $v \in I_p$.

In the architecture A^f, none of the processes is allowed to read the values of the original input variables in I and thus, a routing strategy for process p can only transmit the value of the faulty-input variable f_p to the other processes.

Specification Transformation. The relation between the correct and the faulty input in normal execution and after the occurrence of a fault is established by an assumption on the environment introduced in the specification: While the tuple of values given by the environment to the input variables of a process p and the value given to the corresponding faulty-input variable for that process are constrained to be the same during the normal execution of process p, during the time process p is faulty this may not be the case. Formally, the assumption is $\mathit{faulty\text{-}input} = \mathsf{G} \bigwedge_{p \in P, v \in I_p} (n_p = 0 \rightarrow f_p^v = v)$. Then the formula Φ^f is obtained by substituting in the fault-tolerance specification Φ^t each occurrence of $\mathsf{A}\theta$ by $\mathsf{A}(\mathit{faulty\text{-}input} \rightarrow \theta)$ and each occurrence of $\mathsf{E}\theta$ by $\mathsf{E}(\mathit{faulty\text{-}input} \wedge \theta)$.

For implementations for the architecture A^f we assume that faults do not affect the forwarding of external input by a faulty process. This results in the fault computation tree $\mathcal{FCT}^f(\hat{s}) = \langle T, dir \rangle$, where $T \subseteq D(V^f)^*$ is the greatest total tree such that for all $\sigma \in D(V^f)^*$ and all $d \in D(V^f)$, if $\sigma \cdot d \in T$, then $\sigma \in T$ and *for every* $p \in P$ it holds that $d\langle c_p \rangle = s_p(\sigma\langle In^f(p)\rangle)\langle c_p \rangle$ and if $dir(\sigma)\langle n_p \rangle = 0$ then $d\langle Out(p)\rangle = s_p(\sigma\langle In^f(p)\rangle)\langle Out(p)\rangle$. The following theorem establishes the connection between the fault computation trees for the implementations for the architecture A^f and the implementations for the original architecture A.

Theorem 2. *There exists a finite-state distributed implementation $\hat{s}$ with simple routing for the architecture A that is consistent w.r.t. faults and such that the fault computation tree $\mathcal{FCT}(\hat{s})$ is a model of Φ^t iff there exist a finite-state distributed implementation $\hat{s}^f$ with simple routing for the architecture A^f such that the fault computation tree $\mathcal{FCT}^f(\hat{s}^f)$ is a model of Φ^f.*

Proof (Idea). We define mappings from local input histories for a process $p \in P$ in the architecture A to local input histories for p in A^f and vice versa. An element of $D(In(p))^*$ is mapped to an element of $D(In^f(p))^*$, where the value

of f_p in a state of the image prefix is the value of c_p from the next state in the original prefix, if such a state exists, and is equal to the tuple of values for I_p in the corresponding state of the original prefix otherwise. When mapping $D(In^f(p))^*$ to $D(In(p))^*$, the value of a variable $v \in I_p$ in a state of the image prefix is the same as the value of c_q^v from the next state in the original prefix, if such a state exists and there exists a process q with $v \in I_q$ that is not faulty in the corresponding state of the original prefix, and is the value of f_p^v from the corresponding state in the original prefix otherwise. Based on these mappings we define the respective strategies and show that they have the required properties. The formal definitions and proof can be found in the full version of this paper.

5 From Fault Input-Output Trees to Full Trees

We now present a modification of the classical construction that transforms an automaton on total trees into one that accepts full trees. Our construction accounts for the shape of the total tree resulting from the occurrences of faults.

Consider the architecture A^f and the formula Φ^f obtained as described in Sect. 4 from the architecture A and the fault-tolerance specification Φ^t.

For a global input-output function g for A^f, we define a *fault input-output tree* $\mathcal{FOT}(g) = \langle T, dir\rangle$ similarly to fault computation trees: $T \subseteq D(Ext^f)^*$ is the greatest total tree such that for all $\sigma \in D(Ext^f)^*$ and all $d \in D(Ext^f)$, if $\sigma \cdot d \in T$, then $\sigma \in T$ and for all $p \in P$ with $dir(\sigma)\langle n_p\rangle = 0$ it holds that $d\langle O_p\rangle = g(\sigma\langle I^f\rangle)\langle O_p\rangle$. The tree $\mathcal{FOT}(g)$, which is a total $D(Ext^f)$-labeled $D(Ext^f)$-tree, can be represented as a full $D(Ext^f) \times D(O)$-labeled $D(Ext^f)$-tree where the nodes are labeled additionally with the output of g that determines the enabled directions. Given a full $D(Ext^f) \times D(O)$-labeled $D(Ext^f)$-tree $\mathcal{T}$, we can determine the corresponding total *characteristic tree under faults*, $char_F(\mathcal{T})$.

Definition 4. *Let $\langle D(Ext^f)^*, l\rangle$ be a full $D(Ext^f) \times D(O)$-labeled $D(Ext^f)$-tree. We define* the characteristic tree under faults *as the total $D(Ext^f)$-labeled $D(Ext^f)$-tree $\langle T, dir\rangle = char_F(\langle D(Ext^f)^*, l\rangle)$ as follows: $T \subseteq D(Ext^f)^*$ is the greatest total tree such that for all $\sigma \in D(Ext^f)^*$ and all $d' \in D(Ext^f)$, if $\sigma \cdot d' \in T$, then $\sigma \in T$ and the condition $(*)$ below holds:*
$(*)$ for every $p \in P$, if $d\langle n_p\rangle = 0$, then $d'\langle O_p\rangle = d_o\langle O_p\rangle$, where $l(\sigma) = \langle d, d_o\rangle$.

For the CTL* formula Φ^f, we can construct [11] a symmetric parity automaton S_Φ that accepts exactly the models of Φ^f. This automaton runs on total $D(Ext^f)$-labeled trees, has $2^{O(|\Phi^f|)}$ states and five colors. Since automata transformations are simpler for automata running on full trees, from the symmetric automaton $\mathcal{S}_\Phi$ we construct the alternating parity automaton $\mathcal{A}_\Phi$ that accepts a full tree iff its characteristic tree under faults is a model of Φ^f. The transition function of $\mathcal{A}_\Phi$ uses the information about enabled successors given in the labels: Where $\mathcal{S}_\Phi$ sends a copy to all successors (some successor), $\mathcal{A}_\Phi$ sends a copy to all enabled successors (some enabled successor). However, here the enabled successors are determined only according to the output of the processes that are non-faulty according to the node's label.

Theorem 3. *If $\mathcal{S}_\Phi$ is a symmetric automaton over $D(Ext^f)$-labeled $D(Ext^f)$-trees, we can construct an alternating parity automaton $\mathcal{A}_\Phi$ such that $\mathcal{A}_\Phi$ accepts a $D(Ext^f) \times D(O)$-labeled $D(Ext^f)$-tree $\mathcal{T}$ iff $\mathcal{S}_\Phi$ accepts $char_F(\mathcal{T})$. The automaton $\mathcal{A}_\Phi$ has the same state space and acceptance condition as $\mathcal{S}_\Phi$.*

6 Synthesis of Fault-Tolerant Systems

We reduced the fault-tolerant synthesis problem for the architecture A and the formula Φ^t to the corresponding problem for the architecture A^f and Φ^f. In A^f we can assume that faults do not affect the routing of input and therefore we can reduce the problem to finding delay-compatible input-output functions.

Delay-Compatible Global Input-Output Functions for A^f. From a symmetric automaton $\mathcal{S}_\Phi$ that accepts exactly the total $D(Ext^f)$-labeled $D(Ext^f)$-trees that are models of Φ^f, we construct, as explained in the previous section, an alternating parity tree automaton $\mathcal{A}_\Phi$ that accepts a full $D(Ext^f) \times D(O)$-labeled $D(Ext^f)$-tree $\mathcal{T}$ iff $char_F(\mathcal{T})$ is a model of Φ^f. Via standard transformations we obtain from $\mathcal{A}_\Phi$ a nondeterministic automaton $\mathcal{N}_\Phi$ with number of states doubly exponential in the size of Φ^f that accepts a $D(O)$-labeled $D(I^f \cup N)$-tree $\mathcal{T}$ iff $\mathcal{T}$ corresponds to a global input-output function for A whose fault input-output tree is a model of Φ^f. Via a construction similar to the one in [8], we transform $\mathcal{N}_\Phi$ into a nondeterministic tree automaton $\mathcal{D}_\Phi$ that accepts exactly the labeled trees accepted by $\mathcal{N}_\Phi$ that correspond to delay-compatible global input-output functions. The size of $\mathcal{D}_\Phi$ is linear in the size of $\mathcal{N}_\Phi$. If the language of $\mathcal{D}_\Phi$ is nonempty the nonemptiness test produces a finite-state delay-compatible global input-output function $g : D(I^f \cup N)^* \to D(O)$ for A^f for which $\mathcal{FOT}(g) \models \Phi^f$.

From Global Input-Output Functions to Distributed Implementations. By projecting a finite-state delay-compatible global input-output function g for A^f on the sets O_p of variables we obtain a set of delay-compatible input-output functions $(g_p)_{p\in P}$ for the processes in A^f. These functions are represented as finite automata, which have the same set of states Q_g, which are labeled by elements of $D(O_p)$, and the same deterministic transition function δ_g. For each variable $t_p \in C$, we define $D(t_p) = Q_g \cup \{\bot\}$, $d_0(t_p) = \bot$ and for each $c_p \in C$ we have $D(c_p) = D(I_p^f)$. Then, we define a simple routing $\hat{r} = (r_p)_{p\in P}$ as follows. For every process p and $d \in D(In^f(p))$, the value assigned to t_p by r_p is determined as follows. If $d\langle t_q\rangle = \bot$ for all q, this value is q_g^0 (the initial state). Otherwise, the value is $\delta_g(t, d')$, where $t = d\langle t_q\rangle$ for some process q with $d\langle m_q\rangle = 0$ if one exists, or some fixed process q otherwise, and for every $f_q \in I^f$, $d'\langle f_q\rangle = d\langle c_q\rangle$.

Combining the functions $(g_p)_{p\in P}$ with this simple routing we obtain a finite-state distributed implementation $\hat{s}^f = (s_p^f)_{p\in P}$ for A^f. If the tree $\mathcal{FOT}(g)$ is a model of Φ^f, then so is the tree $\mathcal{FCT}^f(\hat{s}^f)$.

Clearly, vice versa, if there exists a distributed implementation $\hat{s}$ for A^f with $\mathcal{FCT}^f(\hat{s}^f) \models \Phi^f$, then there exists a delay-compatible global input-output function g with $\mathcal{FOT}(g) \models \Phi^f$ and hence the language of $\mathcal{D}_\Phi$ is not empty.

Recalling the relation between the implementations in the architectures A and A^f we established in Sect. 4, we obtain the following result.

Theorem 4. *The fault-tolerant distributed synthesis problem is* 2EXPTIME-*complete for fully connected architectures and external specifications.*

7 Conclusion

We have presented a synthesis algorithm that determines for a fully connected architecture and a temporal specification whether a fault-tolerant implementation exists, and, in case the answer is positive, automatically derives such an implementation. We demonstrated that the framework of incomplete information is well-suited for encoding the effects of faults on the informedness of individual processes in a distributed system. This allowed us to reduce the fault-tolerant distributed synthesis problem to single-process synthesis with incomplete information for a modified specification. We thus showed that the fault-tolerance synthesis problem is decidable and no more expensive than standard synthesis. Establishing general decidability criteria for architectures that are not fully connected as well as extending the scope to broader fault types such as Byzantine faults are two open problems that deserve further study.

References

[1] P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Trans. Program. Lang. Syst.*, 26(1):125–185, 2004.

[2] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Distributed synthesis of fault-tolerant programs in the high atomicity model. In *Proc. SSS*, pages 21–36, 2007.

[3] D. Dolev and H. R. Strong. A simple model for agreement in distributed systems. In *Proceedings of the Asilomar Workshop on Fault-Tolerant Distributed Computing*, pages 42–50, London, UK, 1990. Springer-Verlag.

[4] A. Ebnenasir, S. S. Kulkarni, and A. Arora. FTSyn: a framework for automatic synthesis of fault-tolerance. *STTT*, 10(5):455–471, 2008.

[5] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

[6] B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Proc. LICS'05*, pages 321–330, June 2005.

[7] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *Proc. TACAS*, pages 315–331, 2008.

[8] P. Gastin, N. Sznajder, and M. Zeitoun. Distributed synthesis for well-connected architectures. In *Proc. FSTTCS*, volume 4337 of *LNCS*, pages 321–332, 2006.

[9] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. In *Distributed Systems (2nd edition)*. Addison-Wesley, 1993.

[10] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, 2000.

[11] O. Kupferman and M. Y. Vardi. Church's problem revisited. *Bulletin of Symbolic Logic*, 5(2):245–263, 1999.

[12] P. Manolios and R. Trefler. Safety and liveness in branching time. In *Proc. LICS*, pages 366–374. IEEE Computer Society, 2001.

[13] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. FOCS*, volume II, pages 746–757. IEEE, 1990.