Automatic Synthesis of Distributed Protocols

Rajeev Alur* Stavros Tripakis†

1 Introduction

Protocols for coordination among concurrent processes are an essential component of modern multiprocessor and distributed systems. The multitude of behaviors arising due to asynchronous concurrency makes the design of such protocols difficult, and consequently analyzing such protocols has been a central theme of research in formal verification for decades [25, 35, 13, 32]. Sustained research in improving verification tools has resulted in powerful heuristics for coping with the computational intractability of problems such as Boolean satisfiability and search through the state-space of concurrent systems [11, 26, 15]. Now that automated verification tools are mature enough to be applied to debugging of real-world protocols [12, 33, 23], the new research frontier is *protocol synthesis* for simplifying the design process via more intuitive programming abstractions for specifying the desired behavior.

Traditionally a distributed protocol is modeled as a set of communicating finite-state processes. The correctness is specified by both *safety* and *liveness* requirements. In *model checking*, a given model of the distributed protocol is checked against its correctness requirements specified in *temporal logic*. In *reactive synthesis*, the goal is to automatically derive a protocol from the given logical requirements. The synthesis problem for reactive systems goes back to work in the 1960's [10], with finite automata on infinite words and trees providing the crucial algorithmic apparatus, with some recent efforts to translate these results into practice [39, 30, 8] (see [18] for an excellent survey of the theory of reactive synthesis, and `www.syntcomp.org` for benchmarks and a competition of solvers). However, if we require the implementation to be distributed, then reactive synthesis is undecidable [40, 21]. An alternative, and potentially more feasible approach inspired by *program sketching* [49, 48], is to ask the programmer to specify an *incomplete* protocol to be completed by the synthesizer so as to satisfy all the correctness requirements. This methodology for protocol specification can be viewed as a fruitful collaboration between the designer and the synthesis tool: the programmer has to describe the structure of the desired protocol, but some details that the programmer is unsure about, for instance, regarding corner cases and handling of unexpected messages, are filled in automatically by the tool.

The protocol synthesis problem then reduces to the following *protocol completion* problem: given a set of finite-state machines for communicating processes with incomplete transition functions, given a model of the environment, and given a set of safety and liveness requirements, find a completion of the FSMs for the processes such that the composition satisfies all the requirements. The computational complexity of this problem is PSPACE, the same as that of model checking of distributed protocols. However, now we need to cope with a search with two nested exponentials: the number of possible completions of the incomplete input model is exponential and so is the number of states of the product of all the component processes for any given completion. Advances in model checking offer a way of dealing with the latter, while *counterexample-guided inductive synthesis* (CEGIS) is a new technology that is a potential solution for the former [49, 3, 47]. The synthesis algorithm then consists of iterative invocations of two phases: the *learner* chooses a candidate completion, which is then checked with respect to correctness requirements by the *verifier*; violations of the requirements are supplied to the learner to prune the search space in subsequent iterations.

*Department of Computer and Information Science, University of Pennsylvania. Email: alur@cis.upenn.edu

†Department of Computer Science, Aalto University, and Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. Email: stavros.tripakis@gmail.com

In this paper, using the Alternating Bit Protocol (ABP) as an illustrative example, we explain the formalization of the protocol synthesis problem and review the CEGIS-based algorithm for protocol completion from [6]. Section 2 describes the formal model for finite-state machines communicating via message passing since the quality of the synthesized protocols is very sensitive to the nuances of modeling. In section 3, we formalize the analysis problems of verification, synthesis, and completion for distributed protocols and review the computational complexity of solving these problems. Section 4 contains the detailed model of the ABP example. In section 5, we describe the protocol completion algorithm, along with the results of applying it to the ABP example. Section 6 concludes with a survey of related approaches, insights gained from our work, and directions for future research.

2 Formal Model

The usefulness of automatic synthesis crucially depends on the nuances of the underlying protocol model and how the synthesis problem is formalized. Our formal model is similar to the well-known model of I/O automata for asynchronous distributed protocols [35], but we require the protocol components to be synthesized to be deterministic (sequential) processes. The specific details of the formalization are a minor variation of the description in [6].

2.1 Modeling Protocols

Finite-State Input-Output Processes

A *finite-state input-output process* is a tuple $P = (I, O, Q, Q_0, T, T_f)$ where (1) I is a finite set of *input events*, (2) O is a finite set of *output events* with $I \cap O = \emptyset$ and $I \cup O \neq \emptyset$, (3) Q is a finite set of *states*, (4) $Q_0 \subseteq Q$ is the set of *initial states*, (5) $T \subseteq Q \times (I \cup O) \times Q$ is the *transition relation*, and (6) $T_f \subseteq T$ is the subset of transitions required to be executed in a *strongly fair* fashion. We write a transition $(q, x, q') \in T$ also as $q \xrightarrow{x} q'$. When $x \in I$ (resp., $x \in O$), the transition is called an *input (resp., output) transition* and is also written as $q \xrightarrow{x?} q'$ (resp., $q \xrightarrow{x!} q'$).

Note that we have not explicitly modeled internal transitions: such transitions are useful for constraining what transitions are visible to the environment of a component, but are not crucial for our purpose. On the other hand, fairness assumptions are essential for a protocol to satisfy liveness requirements, and we use the standard notion of strong fairness formalized in the sequel.

A state q is called a *deadlock* if it has no outgoing transitions. A state q is *input-enabled* if for every input event $x \in I$, there exists a state q' such that $q \xrightarrow{x?} q'$. Thus, the process cannot proceed from a deadlock state, and is ready to accept every possible input in an input-enabled state.

Semantics of Processes

Consider a process $P = (I, O, Q, Q_0, T, T_f)$. A *run* of P is a finite or infinite sequence of transitions starting from some initial state: $q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} q_2 \xrightarrow{x_3} \cdots$, with $q_0 \in Q_0$. We call the corresponding sequence of events, $x_1, x_2, x_3, \cdots$, a *trace*.

A state q is called *reachable* if there exists a finite run starting from some initial state and reaching that state: $q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_1} \cdots \xrightarrow{x_n} q$, for some $q_0 \in Q_0$ and some integer $n \geq 0$. The process P is *deadlock-free* if it has no reachable deadlock state.

To verify safety requirements, we need to consider all finite runs of the process, while to verify liveness requirements we should focus on traces corresponding to all *fair* infinite runs. The intuition is that an infinite run is unfair if some transition in T_f is enabled infinitely often, but never taken. An infinite run ρ is said to be *unfair* if there exists a transition $(q, x, q') \in T_f$ such that (q, x, q') never appears in ρ, and the state q appears infinitely often in ρ. Otherwise, the infinite run ρ is said to be *fair*. Note that if T_f is empty, then all infinite runs are fair by definition.

Composition of Processes

We define an asynchronous (interleaving-based) parallel composition operator with rendezvous synchronization. Consider two processes $P_1 = (I_1, O_1, Q_1, Q_0^1, T_1, T_f^1)$ and $P_2 = (I_2, O_2, Q_2, Q_0^2, T_2, T_f^2)$. In order for the composition of P_1 and P_2 to be defined, we require that the processes have no common output events, i.e., $O_1 \cap O_2 = \emptyset$. Then, the *composition of* P_1 *and* P_2, denoted $P_1 \| P_2$, is defined to be the so-called *product* process:

$$P_1 \| P_2 \;\widehat{=}\; \big((I_1 \cup I_2) \setminus (O_1 \cup O_2),\; O_1 \cup O_2,\; Q_1 \times Q_2,\; Q_0^1 \times Q_0^2,\; T,\; T_f\big)$$

where $\big((q_1, q_2), x, (q_1', q_2')\big) \in T$ iff for $i = 1, 2$, if $x \in I_i \cup O_i$ then $(q_i, x, q_i') \in T_i$ else $q_i' = q_i$. The set of fair transitions of the product process is

$$T_f = \{\big((q_1, q_2), x, (q_1', q_2')\big) \in T \mid (q_1, x, q_1') \in T_f^1 \text{ or } (q_2, x, q_2') \in T_f^2\}.$$

Let us provide some intuition for the definition of composition. The set of input events of $P_1 \| P_2$ is $(I_1 \cup I_2) \setminus (O_1 \cup O_2)$, meaning that it contains all inputs of either process which are not outputs of the other. The output events are $O_1 \cup O_2$, i.e., the outputs of either process. This means that an output remains an output even after it has been "matched" by an input. However, an input is removed once it has been matched by an output. Preserving output events allows *multicasting* in the sense that a single output can synchronize with many inputs, from multiple receiver processes.

As is standard, a state of the product process is a pair of states, one from each of its component processes. An initial state of the product is a pair of initial states, one from each component process. A transition $(q_1, q_2) \xrightarrow{x} (q_1', q_2')$ of $P_1 \| P_2$ is one of the following three kinds. One where P_1 issues an output, i.e., $x \in O_1$. In that case, either x is an input of P_2, i.e., $x \in I_2$, or it is not. If $x \notin I_2$, then P_2 does not move, and only P_1 makes a transition. If $x \in I_2$, then the two processes synchronize, i.e., P_1 makes an output transition $q_1 \xrightarrow{x!} q_1'$ while P_2 makes an input transition $q_2 \xrightarrow{x?} q_2'$. The symmetric case is where P_2 issues the output. The third case is where x is an input for $P_1 \| P_2$, i.e., $x \in (I_1 \cup I_2) \setminus (O_1 \cup O_2)$. In this case, if x is an input for both processes, i.e., $x \in I_1 \cap I_2$, then the two processes must synchronize. Otherwise, only one of the two processes moves.

Lastly, the definition of fair transitions for the product ensures that an infinite run of $P_1 \| P_2$ is unfair iff it violates the fairness conditions of either P_1 or P_2. In this way, the fairness assumptions of $P_1 \| P_2$ correspond logically to the conjunction of the fairness assumptions of each of P_1 and P_2.

The composition operator $\|$ is *commutative* and *associative*, and thus, when we need to compose several processes, the order of compositions does not matter.

Deterministic Processes

A distributed system is modeled as a composition of processes, some of which are *protocol processes* that model components of the distributed protocol, and some of which are *environment processes* that capture the environment in which the protocol operates. While environment processes are unrestricted processes, we want protocol processes to satisfy some additional requirements.

First, we want a protocol process to be able to accept any given sequence of inputs. Without such an assumption, one may get solutions to the synthesis problem that "cheat", that is, protocols that achieve certain properties by blocking certain events. For example, in the case of the ABP example, presented in Section 4, the synthesized ABP Sender might achieve the property "every *send* is eventually followed by a *deliver*" by simply refusing to accept *send* events from the Sending Client process. Then, a *send* never happens and the property is satisfied trivially. This requirement, typically referred to as input-enabledness or input-receptiveness, has been formalized in different ways in the literature (see [22, 2, 5, 14, 54, 41]).

Second, we want a protocol process to be implementable as a deterministic sequential program. This means that no state should have two outgoing transitions labeled with the same event: while such nondeterministic transitions are useful to model an environment process (for instance, to specify that a message may or may not get lost), in a deterministic process the next state is determined uniquely from the previous state and the processed event. Furthermore, if an output event is possible in a state, then the process is

committed to producing this output, and cannot accept inputs or produce any other output. In other words, execution of an output transition is not in a "race" with other transitions, and the process can continue only after producing this output. This assumption is common in deterministic models of concurrency such as Kahn process networks [27] and their various dataflow restrictions [34].

Formally, a *deterministic (sequential) process* is a process $P = (I, O, Q, Q_0, T, T_f)$ satisfying all following conditions: (1) the initial state is unique: the set Q_0 contains a single state q_0, (2) the transition relation is deterministic: for every state q and input/output event x, if $q \xrightarrow{x} q_1$ and $q \xrightarrow{x} q_2$ are two transitions, then $q_1 = q_2$, (3) there is no race between input and output transitions: Q is partitioned in two disjoint subsets, the set Q_I of states whose outgoing transitions are all input transitions, called *input states*, and the set Q_O of states with only output outgoing transitions, called *output states*, (4) input states are input-enabled: if $q \in Q_I$, then for every input event x there exists a (unique) state q' such that $q \xrightarrow{x?} q'$, (5) outputs are unique: an output state has a single outgoing (output) transition, (6) inputs are eventually enabled: if the set of input events is non-empty, then from each state $q \in Q$, some input state must be reachable, and (7) enabled output transitions are eventually executed: the set T_f of strongly fair transitions equals the set $T^O = \{q \xrightarrow{x} q' \in T \mid x \in O\}$ of all output transitions.

The first condition ensures that the initial state of a deterministic process is determined uniquely. Suppose the set I of inputs is non-empty. Then, at any step, the process is in either an input state or an output state (due to condition (3)). In an input state, it can only accept inputs, is willing to accept all possible inputs (due to condition (4)), and once such an input is received, the next state is determined uniquely (due to condition (2)). In an output state, the process is not willing to accept any inputs and is ready to produce a unique output event (due to condition (5)) and continue to a uniquely determined next state. Condition (6) ensures that there are no deadlocks and no cycles that consist of only output states, so the process will eventually proceed to a state where it is willing to accept inputs. Finally, we require all output transitions to be strongly fair (condition (7)). Requiring such output-fairness is reasonable, since when a protocol process reaches an output state, we would like the (unique) output transition from that state to be eventually executed. Otherwise, the process is ignored forever, which is clearly unfair. If the set I of inputs is empty, then a deterministic process has only one run consisting of only output transitions, which could be finite terminating in a deadlock state or infinite repeating a cycle of output transitions.

Our definition ensures that a deterministic process has a unique response to any given sequence of input events: if A is a deterministic process and ρ is an infinite sequence of input events, then there is exactly one run $q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} q_2 \xrightarrow{x_3} \cdots$ such that the run is fair and the projection of the trace $x_1, x_2, \ldots$ on the input events equals ρ.

2.2 Modeling Requirements

For a distributed protocol specified as a composition of processes, its semantics is the set of traces capturing its observable behaviors. A requirement is a classification of all possible traces into correct and incorrect, and a protocol meets the requirement if all its traces are correct. Such requirements are often specified in high-level formalisms such as *temporal logic* [36]. We will use instead the more "low-level" formalism of *monitors*. Monitors are extensions of processes, and therefore are easier to present while avoiding a discussion of temporal logic. Moreover, monitors naturally compose with processes using the same principles. Finally, formulas in a temporal logic such as LTL (Linear Temporal Logic) can be translated into the type of monitors used here [56], which means that we do not lose in expressive power.

Automata

Monitors are essentially automata, i.e., processes equipped with additional sets of *accepting states* modeling different types of acceptance conditions. In this work we consider two types of acceptance conditions: *error states* to capture safety properties, and accepting states of type *Büchi* to capture liveness properties.

Formally, we use the term *automaton* for a triple (P, Q_e, Q_a), where $P = (I, O, Q, Q_0, T, T_f)$ is a process, $Q_e \subseteq Q$ is a (possibly empty) set of *error states*, and $Q_a \subseteq Q$ is a (possibly empty) set of *accepting states*.

We require that $Q_e \cap Q_a = \emptyset$.

Monitors

A *monitor* is an automaton (P, Q_e, Q_a) satisfying the following conditions: (1) $P = (I, \emptyset, Q, Q_0, T, \emptyset)$, that is, a monitor has no output events, and no fairness constraints; and (2) every state in Q is input-enabled. These conditions ensure that the monitor is "passive", i.e., when composed with the system being monitored, a monitor only observes but does not otherwise interfere with the system. In particular, input-enabledness ensures that monitors do not block the output events of the system processes they synchronize with. If $Q_e \neq \emptyset$ and $Q_a = \emptyset$ then the monitor is called a *safety monitor*. If $Q_a \neq \emptyset$ and $Q_e = \emptyset$ then the monitor is called a *liveness monitor*.

We use monitors to capture the *negation* of the properties that we want the system to satisfy, i.e., to capture the *violating* traces. A run leading to an error state corresponds to a trace violating a safety property, while a run visiting an accepting state infinitely often corresponds to a trace violating a liveness property. A correct system will be one having no violating traces.

Consider a monitor $M = (P, Q_e, Q_a)$. The notions of run, reachable state, deadlock, and so on, apply to M in the sense that they refer to its corresponding process $P = (I, O, Q, Q_0, T, T_f)$.

The monitor M is said to be *safe* if it has no reachable error states, i.e., no $q \in Q_e$ is reachable.

An infinite run of M is said to be *accepting* if it visits accepting states (i.e., states in Q_a) infinitely often. The monitor M is said to be *live* if it has no infinite run that is both fair and accepting.

Composition of Monitors, Automata, and Processes

We next define automata composition as an extension of process composition. Since monitors are special cases of automata, this also defines composition of monitors, as well as composition of monitors with automata. Moreover, processes can be viewed as special cases of automata with empty sets of error and accepting states. Therefore, the composition of all three types of components, processes, automata, and monitors, is also defined. Associativity and commutativity of process composition extends to the case of automata composition as well.

Consider two automata $A_1 = (P_1, Q_e^1, Q_a^1)$ and $A_2 = (P_2, Q_e^2, Q_a^2)$. In order for the composition of A_1 and A_2 to be defined, we require that the composition of their processes, $P_1 \| P_2$, is defined. Let $P_1 = (I_1, O_1, Q_1, Q_0^1, T_1, T_f^1)$, $P_2 = (I_2, O_2, Q_2, Q_0^2, T_2, T_f^2)$, and $P_1 \| P_2 = (I, O, Q, Q_0, T, T_f)$. Then, the *composition of* A_1 *and* A_2, denoted $A_1 \| A_2$, is defined to be the automaton $A_1 \| A_2 = (P_1 \| P_2,\ Q_e,\ Q_a)$, where $Q_e = (Q_e^1 \times Q_2) \cup (Q_1 \times Q_e^2)$ and $Q_a = (Q_a^1 \times Q_2) \cup (Q_1 \times Q_a^2)$.

A state (q_1, q_2) is an error state of $A_1 \| A_2$ if either q_1 is an error state of A_1 or q_2 is an error state of A_2. Similarly, (q_1, q_2) is an accepting state of $A_1 \| A_2$ if either q_1 is an accepting state of A_1 or q_2 is an accepting state of A_2. When A_1 and A_2 are monitors, these definitions imply that in order for a violation to occur in the product monitor, it must occur in at least one of its component monitors. Note that this definition "works" because we use monitors to model not the properties we want the system to satisfy, but the *negation* of those properties. Indeed, suppose we want the system to satisfy several properties, say $\phi_1, \phi_2, ..., \phi_n$; that is, we want the system to satisfy their *conjunction* $\phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$. Then we can build a separate monitor M_i which captures the negation of each property, i.e., $\neg\phi_i$. The definition of product error and accepting states ensures that the product $M_1 \| \cdots \| M_n$ captures the *disjunction* $\neg\phi_1 \vee \cdots \vee \neg\phi_n$, which is equivalent to the negation of the global property ϕ.

3 Analysis and Synthesis Problems

3.1 Protocol Verification

In the verification problem, the protocol is modeled as a set $P_1, \ldots P_m$ of processes and the requirements are captured by a set $M_1, \ldots M_n$ of monitors. The verification problem is to check if the system is correct:

Problem 1 (Distributed Protocol Verification) *Given a set of processes $P_1, \ldots P_m$ and a set of monitors $M_1, \ldots M_n$, check if (1) the product $P_1 \| \cdots \| P_m$ is deadlock-free, and (2) for $i = 1, \ldots n$, the product $P_1 \| \cdots \| P_m \| M_i$ is both safe and live.*

The first condition checks the implicit requirement of absence of deadlocks and the second condition checks that the protocol satisfies each of the explicit safety and liveness requirements. Each of these conditions can be checked separately, and for the second, correctness with respect to each monitor can also be checked separately. Each check requires exploration of the global state-space of the product of all the processes. Checking safety corresponds to checking reachability of error states in the product, while checking liveness corresponds to detecting reachable cycles that contain accepting states and satisfy fairness assumptions. The theoretical complexity of the verification problem is PSPACE since this exploration can be done without explicitly constructing the exponential-sized product graph [56, 13]:

Theorem 1 *The distributed protocol verification problem is* PSPACE*-complete.*

The exponential growth in the size of the global state-space is called "state-space explosion", and in the last thirty years a number of heuristic approaches have been proposed to cope with this problem. Examples of model checkers that incorporate these approaches and have been successfully applied to real-world protocols include SPIN [26], SMV [11], and Murphi [16].

3.2 Protocol Synthesis

In the protocol synthesis problem, we are given a set of processes that model the environment and a set of monitors that capture the requirements. We are also given the communication architecture, that is, the set of inputs and outputs through which the protocol processes to be synthesized may communicate. The synthesis problem then is to construct the desired protocol processes as deterministic processes so that the composed system meets the safety and liveness requirements of the monitors as well as the implicit requirement of absence of deadlocks.

An *input-output interface* (or *IO interface*, for short) is a pair (I, O), where I is a set of input events and O is a set of output events such that $I \cap O = \emptyset$ and $I \cup O \neq \emptyset$. We are now ready to define the *distributed protocol synthesis* problem (DPS):

Problem 2 (Distributed Protocol Synthesis) *Given a set of processes $P^e_1, ..., P^e_k$, called the* environment processes*, a set of IO interfaces $(I_1, O_1), ..., (I_m, O_m)$, and a set of monitors $M_1, ..., M_n$, find, if there exist, a set of deterministic processes $P_1, ..., P_m$, called* protocol processes*, such that: (1) for $i = 1, \ldots m$, each P_i has the interface (I_i, O_i), that is, $P_i = (I_i, O_i, Q_i, q^i_0, T_i, T^i_f)$, for some Q_i, q^i_0, T_i, T^i_f, (2) the product $P^e_1 \| \cdots \| P^e_k \| P_1 \| \cdots \| P_m$ is deadlock-free, and (3) for each $i = 1, \ldots n$, the product $P^e_1 \| \cdots \| P^e_k \| P_1 \| \cdots \| P_m \| M_i$ is both safe and live.*

To understand the difficulty in solving the distributed protocol synthesis problem, let us focus on a special case: suppose we have no environment processes and a single deterministic safety monitor M. Let the set of IO interfaces be $(I_1, O_1), ..., (I_m, O_m)$. That is, we want to synthesize protocol processes A_i with input I_i and outputs O_i, where the monitor M imposes a safety requirement on the desired communication pattern. The synthesis problem then corresponds to finding a winning strategy in a multi-player game played over the states of monitor M, where the players correspond to the processes to be synthesized. The game starts in the initial state of the monitor. At every step one of the players takes a step: a step by the process A_i corresponds to an event in O_i and this updates the monitor state according to the transition function. The strategy of the player A_i determines which event in O_i is to be produced, and it can depend only on the sequence of events in I_i that have been played so far. Such a strategy can be formalized as a function from I^*_i to O_i: based on the sequence of inputs observed so far, it produces the next output. Note that the player does not know the entire history, and thus, has only *partial information* about the state of the monitor. If we fix a strategy for each of the players, then we get a unique run of the monitor, and if this run avoids the error states, then the strategies are winning. Such winning strategies when viewed as deterministic automata give

us a solution to the synthesis question. Thus, the distributed synthesis problem reduces to finding winning strategies in a multi-player partial information game. Such games unfortunately are undecidable even when the number of players is two [38]. This does not directly imply undecidability of the distributed synthesis problem, but essentially the same proof idea can be used to show that as long as we have two unknown processes and a non-trivial specification, the synthesis problem is undecidable [40].

Theorem 2 *The distributed protocol synthesis problem is undecidable.*

There have been efforts to identify restrictions on the pattern of communication among the processes to be synthesized so as to ensure decidability of the synthesis problem. We refer the reader to [21, 18] for sufficient and necessary conditions on the communication architecture for decidability. Unfortunately, the communication pattern in our case study of the Alternating Bit Protocol does not fall within the decidable class.

Supervisory Control

Reactive synthesis is related to the theory of *supervisory control* for *discrete-event systems* [43, 44]. A comparative introduction of reactive synthesis and supervisory control can be found in [17]. Supervisory controller synthesis has been studied extensively, for the cases of fully observable or partially observable systems, centralized or decentralized controllers, and many other cases (e.g., see [50, 9]). Particularly related to the topic of this paper is the case of decentralized control. Undecidability of decentralized supervisory control problems has been shown in [31, 51] for the case of ω-regular languages and in [52, 53] for the case of regular languages. Other variants of decentralized supervisory control had earlier been shown to be decidable [46, 45].

3.3 Protocol Completion

In the protocol synthesis problem, we only know the input-output interface of each of the protocol processes to be synthesized, and need to figure out the set of states and transitions necessary to implement the desired logic. In a less demanding version of the problem, that we call *completion*, we have some partial information regarding the states and transitions of the protocol processes and need to only fill in the missing details.

An *incomplete process* is a process with some of its elements missing, or incomplete. For the purposes of this paper, we will define an incomplete process to be simply a process with a possibly incomplete set of transitions. Extra transitions can be added to such a process during a *completion* process. Formally, an incomplete process P is defined by a tuple of the same type as a normal process, where the set of fairness constraints is initially empty: $P = (I, O, Q, q_0, T, \emptyset)$. Given a set of transitions $T' \subseteq Q \times (I \cup O) \times Q$, the *completion of P with T'* is the new process $P' = (I, O, Q, q_0, T \cup T', T_f)$, which is required to be a deterministic process. The requirement that P' is deterministic implicitly determines the set of strongly fair transitions T_f to be the set of all output transitions in $T \cup T'$. This includes all existing output transitions in T, as well as any newly added output transitions in T'.

We now define a second synthesis problem, which we call *distributed protocol completion* (DPC):

Problem 3 (Distributed Protocol Completion) *Given a set of environment processes $P^e_1, ..., P^e_k$, a set of incomplete protocol processes $P_1, ..., P_m$, and a set of monitors $M_1, ..., M_n$, find, if there exist, sets of transitions $T_1, \ldots T_m$ such that: (1) for each $i = 1, \ldots m$, the completion P'_i of P_i with T_i is a deterministic process, (2) the product $P^e_1 \| \cdots \| P^e_k \| P'_1 \| \cdots \| P'_m$ is deadlock-free, and (3) for each $i = 1, \ldots n$, the product $P^e_1 \| \cdots \| P^e_k \| P'_1 \| \cdots \| P'_m \| M_i$ is both safe and live.*

While protocol synthesis is undecidable, the protocol completion problem is decidable for the following reason. Every incomplete protocol process has a finite number of states, and only transitions, but no states, can be added during completion. The sets of input and output events of each process are both finite, and thus so is the set of all possible transitions. There is a finite number of protocol processes to be completed, and each one admits a finite number of completions, therefore, the total number of completion combinations

is also finite. For every possible completion, we can then use protocol verification to check if the completion satisfies all the desired requirements. It is also easy to see that the protocol completion problem belongs to the complexity class PSPACE: the description of each completed process P_i' is polynomial in the description of the incomplete process P_i, and thus, the desired completions can be guessed in polynomial space, and for a given completion, its validity can be checked using a PSPACE algorithm for the protocol verification problem.

Theorem 3 *The distributed protocol completion problem is* PSPACE*-complete.*

The completion problem in general has the same complexity, PSPACE, as the verification problem, but unlike the verification problem, it is still hard (NP-complete) even for a constant number of processes [6].

4 Illustrative Example: the Alternating Bit Protocol

The Alternating Bit Protocol (ABP) is a standard communication protocol that provides reliable (i.e., *lossless*) transmission of a message over an unreliable (i.e., *lossy*) channel. ABP achieves this by retransmitting the message when the message is deemed lost. ABP is used routinely to illustrate formal modeling and verification techniques [26, 35, 42], and indeed is simple enough to illustrate our approach to distributed protocol synthesis.

4.1 ABP System Architecture

The system architecture of the ABP model is shown in Figure 1. The figure shows all the processes of the model, depicted as rectangles or circles, and their communication events (inputs and outputs), depicted as labeled arrows. The figure also shows the safety and liveness monitors used in our model. Protocol processes are denoted by circles. Environment processes and monitors are denoted by rectangles.

The system contains seven processes in total:

- The protocol processes *ABP Sender* and *ABP Receiver*.
- The environment processes *Forward Channel* and *Backward Channel*. These two processes model the two lossy channels linking the ABP Sender and Receiver.
- The environment processes *Sending Client* and *Receiving Client* which model two client processes: the former wants to send messages to the latter. The messages must be transmitted reliably, even though the channels are unreliable.
- The environment process *Timer* which issues timeouts.

The communication between the system processes is as follows:

- *ABP Sender* communicates with *Forward Channel* via events p_0 and p_1, modeling the transmission of a message annotated with bit 0, and that of a message annotated with bit 1, respectively. We are not interested in the value of the message itself, and therefore do not model what is carried in the "body" of the message, but only in its "header". In this simple protocol, the header consists of a single bit. Events p_0, p_1 are output events for the *ABP Sender*, and input events for the *Forward Channel*.
- *Forward Channel* communicates with *ABP Receiver* via events p_0' and p_1'. These two events also model the transmission of a message with 0 or 1, but they are primed, in order to be distinct from p_0, p_1. The reason we want to distinguish p_0', p_1' from p_0, p_1, is that p_0, p_1 are used to synchronize the transitions of *ABP Sender* and *Forward Channel* (but not of *ABP Receiver*), whereas p_0', p_1' are used to synchronize the transitions of *Forward Channel* and *ABP Receiver*.
- Events a_0, a_1 and a_0', a_1' model *acknowledgment* messages annotated with the bit 0 or 1, and sent from the receiver to the sender through the backward channel.

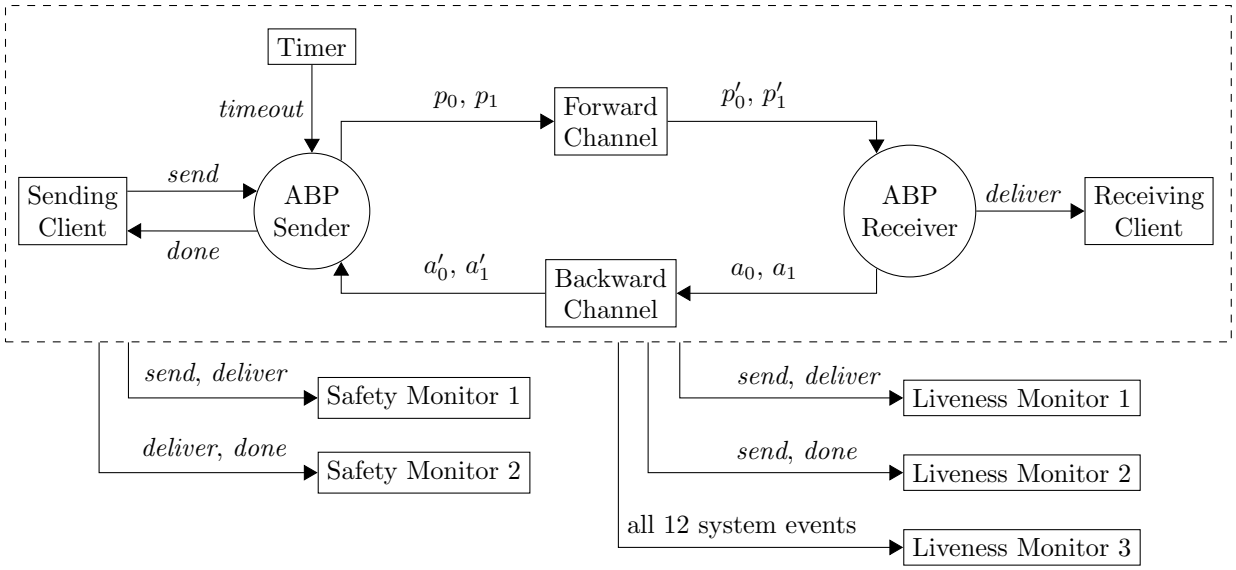


Figure 1: ABP system architecture.

- The *Sending Client* communicates with *ABP Sender* with events *send* and *done*. Event *send* models the message that the sending client gives to the ABP protocol for delivery to the receiving client. Event *done* models the response from the protocol that the message has been successfully delivered.
- Event *deliver* models the delivery of the message to the receiving client.
- Event *timeout* models the occurrence of a timeout. The sender uses such timeouts to retransmit messages as it deems necessary.

4.2 The Environment Processes

The two channel processes are shown in Figure 2. Circles denote states and arrows between states denote transitions. Arrows without a source state denote initial states. Transitions with bold lines and double arrows denote strong fairness constraints, further discussed in §4.5. Transitions are labeled with input or output events. A '?' following an event indicates an input event, while '!' indicates an output event (c.f. §2.1).

Both channels have capacity 1, meaning that they can store at most one message. In the *Forward Channel*, state f_0 corresponds to the channel being empty, either because it hasn't received any message yet, or because it has lost the last message received. When a message, say p_0, is sent to an empty channel, the channel may nondeterministically either take the self-loop transition and remain at f_0, meaning that it loses the message, or take the transition from f_0 to f_1, meaning that it stores the message. Then, from f_1, the channel may choose to take, nondeterministically, either the transition labeled p'_0! back to f_0, or the self-loop with the same label which remains at f_1. The first transition corresponds to the channel forwarding (correctly) a single copy of the message it received. The self-loop transition corresponds to the channel choosing to forward multiple copies (two or more) of the message. This models another typical defect of communication channels, namely, message duplication. In addition to the outgoing transitions from f_1 labeled with the output event p'_0, state f_1 also contains self-loop transitions labeled with the input events p_0 and p_1. These self-loops capture what happens if the channel receives a new message while it still hasn't completed forwarding the last message it received. In such a case, the most recent message is simply ignored, i.e., lost.

The *Backward Channel* is similar to the *Forward Channel* and therefore not described in further detail.

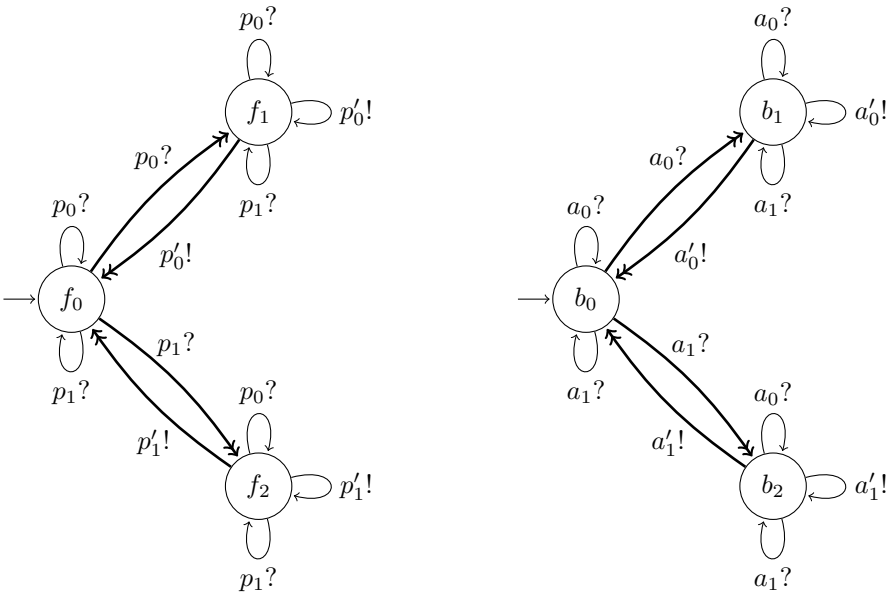


Figure 2: Environment processes *Forward Channel* (left) and *Backward Channel* (right). Transitions in bold lines and double arrows are strongly fair, meaning they cannot be enabled infinitely often without being taken.

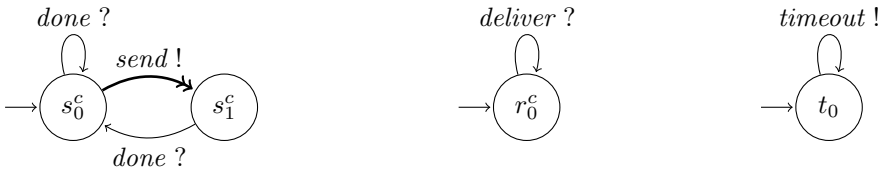


Figure 3: Environment processes *Sending Client* (left), *Receiving Client* (middle), and *Timer* (right).

The environment processes *Sending Client*, *Receiving Client*, and *Timer* are shown in Figure 3. *Sending Client* sends a message and then waits for *done* before sending the next message. *Receiving Client* simply accepts any message delivered to it. *Timer* can issue a timeout at an arbitrary point in time. This is a conservative way of modeling timeouts, which in reality occur at precise moments in time, after certain durations specified as part of the timing parameters of the protocol. Our model is *untimed* and cannot capture such quantitative constraints. Still, our model is conservative: if a protocol works correctly assuming that timeouts can occur at *any* time, then surely it will also work correctly when timeouts can occur only after certain specified durations. *Timed* formalisms such as *timed automata* [4] exist, allowing to capture quantitative timing constraints. But such formalisms typically involve much more computationally expensive analysis and synthesis algorithms.

Note that all environment processes are able to accept all their input messages at every state. This holds trivially for process *Timer* which has no inputs. The self-loop labeled *done*? at state s_0^c of the *Sending Client* is added in order to achieve this property. This property ensures that none of the environment processes can block any output event of another process at any time.

4.3 Safety and Liveness Properties: the Monitors

In addition to deadlock freedom, the system must satisfy certain safety and liveness properties captured by the safety monitors of Figures 4 and 5, and the liveness monitors of Figures 6 and 7 (and possibly also of

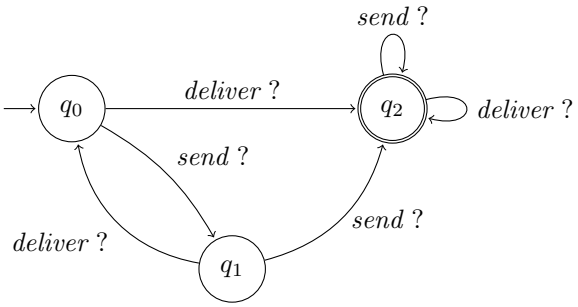


Figure 4: Safety monitor 1 for the ABP system: "*send* and *deliver* happen in the right order". State q_2 is the error state, meaning that the safety property is violated if the monitor ever enters that state.

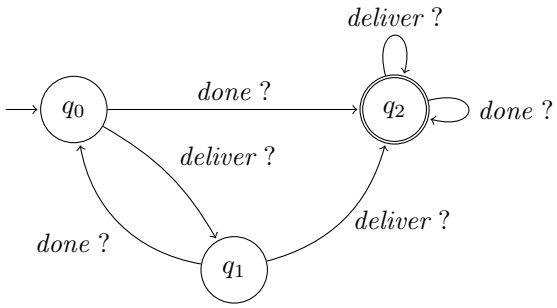


Figure 5: Safety monitor 2 for the ABP system: "*deliver* and *done* happen in the right order".

Figure 8, as discussed in the sequel).

The safety monitor of Figure 4 captures the property that events *send* and *deliver* must occur in the right order: a *deliver* cannot occur unless a *send* occurs before, and two *send*s cannot occur in a row without a *deliver* in-between. State q_2 of the safety monitor is the error state. If ever the safety monitor enters that state, the property has been violated.

The safety monitor of Figure 5 captures the property that events *deliver* and *done* must occur in the right order. This monitor has exactly the same structure as the monitor of Figure 4, except that *deliver* is replaced by *done*, and *send* by *deliver*.

We want the system to satisfy the liveness property that every *send* must eventually be followed by a *deliver*, i.e., that every message is eventually delivered.[1] The liveness monitor of Figure 6, with accepting state q_1, captures the *negation* of this property. That is, a behavior is accepted by this monitor iff it violates the property. Such a violating behavior is one where at some point a *send* occurs (upon which the monitor moves from q_0 to q_1) and no *deliver* ever occurs after that (therefore the monitor gets "stuck" in q_1 forever,

[1] In LTL, this property can be stated as $\mathbf{G}(send \rightarrow \mathbf{F}\, deliver)$.

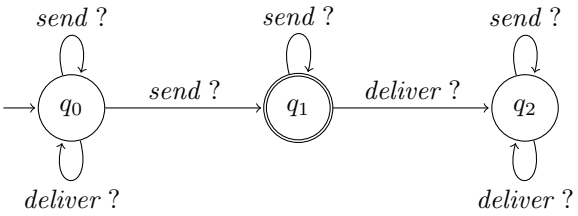


Figure 6: Liveness monitor 1 for the ABP system: "every *send* is eventually followed by a *deliver*".

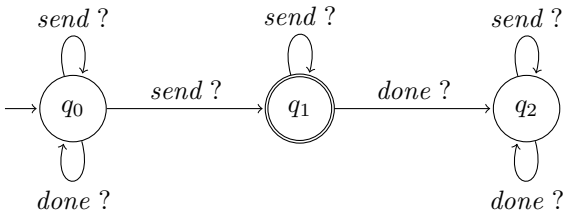


Figure 7: Liveness monitor 2 for the ABP system: " every *send* is eventually followed by a *done* ".

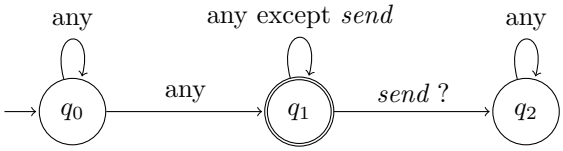


Figure 8: Liveness monitor 3 for the ABP system: " *send* occurs infinitely often ". Each transition labeled "any" represents a multitude of transitions, one for each input event to the monitor. Similarly, the transition labeled "any except *send*" represents a multitude of transitions, one for each input except input *send*.

thus accepting the behavior). Note that this automaton is nondeterministic: from q_0, upon observing event *send*, it can either remain at q_0 or move to q_1. This nondeterminism captures in a simple manner violating behaviors where the *send* event causing the violation is not necessarily the first one.

The liveness monitor of Figure 7 captures the property that every *send* must eventually be followed by a *done*, i.e., that the sending client is eventually notified of the successful delivery of a message.[2] This monitor has exactly the same structure as the monitor of Figure 7, except that *deliver* is replaced by *done*.

A final liveness property that we sometimes use in our experiments is the property that *send* occurs infinitely often.[3] This property is useful for eliminating some synthesis solutions which turn out to be blocking.[4] The liveness monitor for this property is shown in Figure 8. This monitor receives as inputs all 12 events of the ABP system. Each transition labeled "any" represents a multitude of 12 transitions, one for each of these 12 input events. Similarly, the transition labeled "any except *send*" represents 11 transitions, one for each input except input *send*. Note that from state q_0 of this monitor there is a non-deterministic choice for every input event, e.g., for input event p_0, there is the transition $q_0 \xrightarrow{p_0} q_0$ and also the transition $q_0 \xrightarrow{p_0} q_1$. On the other hand determinism holds at states q_1 and q_2. In particular, *send* leads only to q_2 from q_1, and any other event from q_1 leads back to q_1.

4.4 ABP Sender and Receiver

We now present a first version of the ABP sender and receiver processes. This version was built "manually", based on textbook descriptions of the ABP protocol.

ABP Sender

The ABP Sender process is shown in Figure 9. It is a deterministic process with 8 states, out of which 4 are input states (s_0, s_2, s_4, s_6) and 4 are output states (s_1, s_3, s_5, s_7). Notice that all input states have outgoing transitions for each one of the 4 input events of the ABP Sender, therefore, all these states are input-enabled.

[2] In LTL, this property can be stated as $\mathbf{G}(send \rightarrow \mathbf{F}done)$.
[3] In LTL, this property can be stated as $\mathbf{GF}send$.
[4] Such solutions may be generated due to current limitations of our tool, which does not ensure that all conditions of a deterministic sequential process are met. In particular the tool does not ensure condition (6) – c.f. 'Deterministic Processes', page 4.

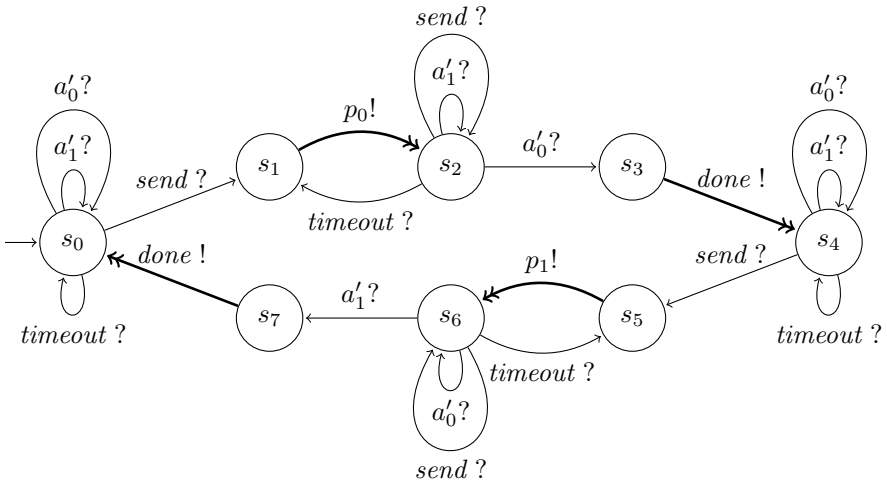


Figure 9: "Manually" constructed ABP Sender.

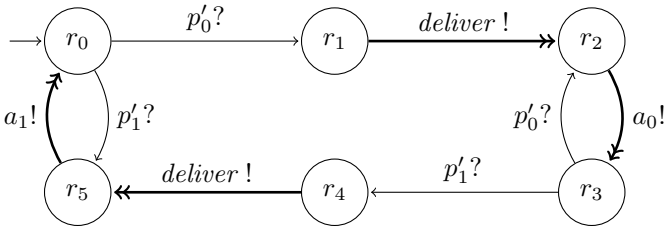


Figure 10: "Manually" constructed ABP Receiver.

The sender starts at state s_0, where it is idling, awaiting a *send*, i.e., awaiting to begin transmission of a message, and ignoring all other input events (a_0', a_1', and *timeout*). Once it receives a *send*, the sender moves to state s_1 and sends p_0 to the forward channel. After that it moves to state s_2, where it waits for an acknowledgment from the ABP Receiver process. The acknowledgment must be annotated with the same bit as the message, i.e., bit 0 in this case. Therefore the expected acknowledgment event at state s_2 is a_0'. If the wrong acknowledgment a_1' is received, it is simply ignored (self-loop at state s_2). If the right acknowledgment a_0' is received, this means that the message has been successfully transmitted: therefore the sender moves to state s_3, upon which it sends a *done* event to the Sending Client, and moves to state s_4 to wait for a new message.

At state s_2, it is also possible for the sender to receive a *timeout*. In that case, it moves back to state s_1 and retransmits the message, i.e., sends a new p_0 event to the forward channel.

The operation of the sender along states s_4, s_5, s_6, s_7 is symmetric to its operation along states s_0, s_1, s_2, s_3. The difference is that now the bit is switched from 0 to 1 (hence the term *alternating bit*).

ABP Receiver

The ABP Receiver process is shown in Figure 10. It is a deterministic process with 6 states, out of which 2 are input states (r_0, r_3) and 4 are output states (r_1, r_2, r_4, r_5). Note that all input states of the ABP Receiver process are input-enabled.

The receiver starts at state r_0, where it is idling and awaiting for an input event. If that input event is p_0', then it indicates a new incoming message, since the alternating bit 0 is the correct "next one" in sequence. In that case, the receiver delivers the message to the Receiving Client by sending a *deliver*, and replies to the ABP Sender with an acknowledgment, by sending a_0 to the Backward Channel. Note that the

acknowledgment is annotated with the same bit as the message received, namely, 0 in this case.

If, on the other hand, the input received at state r_0 is not p'_0, but p'_1, then this indicates an "out of sequence" incoming message. For instance, it could be a redundant retransmission by the sender, due to a lost acknowledgment. In that case, the receiver retransmits the last transmitted acknowledgment, i.e., a_1 in this case.

The operation of the receiver at states s_3, s_4, s_5 is symmetric to that at states s_0, s_1, s_2, with the role of the alternating bit switched from 0 to 1.

4.5 Fairness Assumptions

The ABP Sender and Receiver cannot fulfill the protocol's liveness requirements unless we impose some *fairness* assumptions on the model.

The first assumption that we need is that the channels do not lose all messages. Indeed, without this assumption, the behavior *send*, p_0, *timeout*, p_0, *timeout*, $\cdots$, is possible, where the Forward Channel keeps self-looping at state f_0, always losing message p_0, thus violating the property that *send* is eventually followed by *deliver*. To avoid such behaviors, we declare the following transitions of the two channels as strongly fair: $(f_0,\ p_0?,\ f_1)$, $(f_0,\ p_1?,\ f_2)$, $(b_0,\ a_0?,\ b_1)$, and $(b_0,\ a_1?,\ b_2)$.

The second assumption that we need is that the channels don't get "stuck" at their "full" states, continuously replicating output messages. Here, the violating traces are more subtle. For instance, during the first *send* cycle, where the sender sends p_0, the Forward Channel may get stuck at state f_1. In the next *send* cycle, the sender sends p_1. But since the Forward Channel is at state f_1, it keeps ignoring p_1. Therefore, a *deliver* never follows the second *send*. To avoid such behaviors, we declare additionally the following transitions of the two channels as strongly fair: $(f_1,\ p'_0!,\ f_0)$, $(f_2,\ p'_1!,\ f_0)$, $(b_1,\ a'_0!,\ b_0)$, and $(b_2,\ a'_1!,\ b_0)$.

The third assumption that we need is that if *deliver* is possible, it will eventually happen. This is achieved by the strong fairness assumption on the transitions $(r_1,\ \mathit{deliver}!,\ r2)$ and $(r_4,\ \mathit{deliver}!,\ r5)$ of the ABP Receiver. Without this assumption, the behavior *send*, p_0, *timeout*, p'_0, p_0, *timeout*, p_0, *timeout*, $\cdots$ is possible. In this behavior, the Forward Channel gets "stuck" at state f_1, opting for the self-loop transition there when it sends p'_0, instead of the transition returning to f_0. The ABP Sender keeps timing-out and retransmitting p_0, between states s_1 and s_2. The ABP Receiver is "stuck" at state r_1, waiting to issue a *deliver*. The transitions corresponding to *deliver* are always enabled, but never taken. The fairness assumptions on the *deliver* transitions rule out this kind of behavior.

A final fairness assumption that we may wish to impose is to declare the transition $(s^c_0,\ \mathit{send}!,\ s^c_1)$ of the Sending Client as strongly fair. Without this fairness condition, uninteresting behaviors such as an infinite sequence of *timeout*s become possible. This fairness condition is needed for the "infinitely often *send*" property (liveness monitor 3), but not for the other two liveness properties described above.

Several other transitions of the ABP Sender and Receiver are declared to be strongly fair, in particular, all output transitions of these processes (see Figures 9 and 10). This is done to conform to the default requirement that all output transitions of the deterministic protocol processes be strongly fair. However, these extra fairness assumptions are not strictly necessary, as the assumptions listed above are sufficient in order to satisfy all three liveness properties discussed above.

4.6 ABP as a Solution to a Distributed Protocol Synthesis Problem

ABP can be seen as a solution to the problem of finding a protocol that guarantees reliable transmission of messages over unreliable channels. Formally, the ABP model presented above has no deadlocks, and satisfies the properties expressed by the monitors of Figures 4, 5, 6, 7 and 8.

Moreover, the ABP Sender and Receiver of Figures 9 and 10 are deterministic sequential processes. Therefore, these two protocol processes can be seen as a solution to the Distributed Synthesis Problem 2, where: there are $k = 5$ environment processes – Sending and Receiving Clients, Timer, Forward and Backward Channels; there are $m = 2$ protocol processes – the ABP Sender and Receiver, with interfaces as shown in Figure 1; and there are $n = 5$ monitors – the safety and liveness monitors presented above.

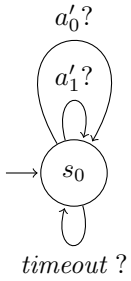


Figure 11: Blocking Sender: it blocks the *send* event of the Sending Client by not having any transition labeled with that event.

It is worth noting that the requirements imposed on deterministic sequential processes, such as input-enabledness of input states, are crucial, in the sense that ignoring these requirements may result in trivial solutions. For instance, the *Blocking Sender* shown in Figure 11, together with the receiver of Figure 10, satisfy all requirements that are satisfied by the correct protocol. But clearly this is not a solution we want, since the Blocking Sender achieves the liveness property "every *send* must be eventually followed by a *deliver*" by simply refusing to accept *send* events, even though *send* is in its input interface. A *send* can therefore never occur, and the above liveness property is trivially satisfied. The same is true with the other liveness property. Requiring input states of protocol processes to be input-enabled eliminates such pathological solutions.

5 Automatic Protocol Completion

5.1 Solving the Distributed Protocol Completion Problem

The total number of completions of given incomplete processes may be finite, but in most realistic examples it is huge. Even in the case of the relatively simple ABP example, there are more than 2 trillion candidate completions (see Section 5.3). Checking each of them for correctness can be done automatically with a model checker. But even if model checking each candidate is fast, the sheer number of candidates makes enumerating and checking all of them an impossible task.

An alternative to brute-force enumeration is proposed in [6]. As we shall see in §5.3, this alternative method allows to complete examples like the ABP in less than a minute. The method can be viewed as an instance of the so-called *counterexample-guided inductive synthesis* paradigm (CEGIS) [49, 48]. At a high-level, the algorithm works by maintaining a set of *completion constraints* that any correct completion must satisfy. The algorithm then repeatedly chooses a candidate completion that satisfies these constraints. If no such completion exists, the algorithm terminates and reports no solution. Otherwise, the chosen completion is checked against the correctness requirements using a model checker. If the chosen completion satisfies the requirements then a solution is found and the algorithm terminates. Otherwise, the model checker returns a *counterexample* showing one possible violation of the requirements. From this counterexample as well as possibly additional knowledge, the synthesis algorithm extracts information used to create more constraints on the set of correct completions, therefore pruning further the search space. As can be seen, this method relies heavily on the counterexamples returned by the model checker, hence the term *counterexample-guided*.

The approach is illustrated in Figure 12. The figure depicts the interplay of the two main components of the algorithm: the *Learner* component which maintains the set of completion constraints and generates the candidate completions; and the *Verifier* component which checks those candidates for correctness. The Learner also processes the counterexamples returned by the Verifier when the candidate completion is incorrect. Apart from the candidate completions, other inputs to the Verifier include the overall model (system architecture, environment processes, incomplete protocol processes, monitors, etc.). Additional inputs to the Learner are the incomplete processes, and possibly also the overall model.

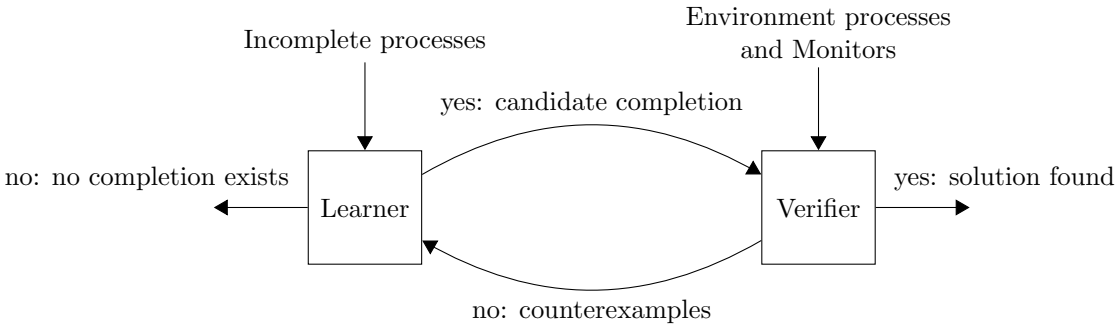


Figure 12: Block diagram of a completion algorithm that uses the CEGIS paradigm.

The completion constraints maintained by the algorithm of [6] are propositional formulas over a set of Boolean variables. There is one such variable for each candidate transition that can be added to each individual incomplete processes. The variable represents whether the transition can be added or not. For example, if variables x_2 and x_7 represent some transitions t_2 and t_7, respectively, then the formula $\neg(x_2 \wedge x_7)$ states that no completion must add both t_2 and t_7.

It is beyond the scope of this paper to explain the algorithm of [6] in detail. But let us give a flavor of how the algorithm works in one simple case. Suppose the algorithm currently explores the candidate set of added transitions $T = \{t_2, t_7\}$. This means that we are attempting to add to the incomplete protocol two transitions, namely t_2, t_7, and we want to check whether this addition is correct. Suppose it is not: suppose the model checker detects a safety violation. This means that the completed protocol has a run reaching an error state. But this implies that any set of added transitions T' such that $T' \supseteq T$, will also be incorrect. Indeed, if a run is possible with a given set of transitions, adding even more transitions cannot eliminate this run. Therefore, if an error state is reachable with T, it will also be reachable with any $T' \supseteq T$. Thus, T' will have the same safety violation as T. This reasoning implies that *any* completion that contains at least t_2 and t_7, and possibly more transitions, is bound to be incorrect. Therefore, we add the constraint $\neg(x_2 \wedge x_7)$ to the set of completion constraints.

5.2 From Scenarios to Incomplete Processes

The incomplete protocol processes used in protocol completion do not necessarily have to be "manually" designed. They can also be generated automatically from example *scenarios*, as proposed in [6]. Let us briefly illustrate this in the context of the ABP example. For a more detailed description of the scenario-based methodology we refer the reader to [6].

An example scenario for the ABP protocol is shown in Figure 13. This scenario is given in the form of a *message sequence chart* (MSC). MSCs are a popular graphical notation for describing distributed protocol interactions. MSCs are also an IEEE standard [1].

In the MSC shown in Figure 13, every vertical dotted line corresponds to the time-line of a process in the system. This scenario involves six processes in total (process *Timer* does not participate in this scenario). Each labeled arrow corresponds to a message sent by one process to another. Although there is no quantitative time in this model, there is an implicit ordering of events: the reception of a message happens after the transmission of the same message; also, within a process, events happen later as we move further down the line. Thus, we can begin "reading" the scenario as follows: first, the *Sending Client* sends message *send* to the *ABP Sender*; the *ABP Sender* receives message *send*, and then sends message p_0 to the *Forward Channel*; the *Forward Channel* receives p_0 and then sends p'_0 to the *ABP Receiver*; etc. It is worth noting that, although the transmission of *deliver* by the *ABP Receiver* happens before its transmission of a_0, there is no guaranteed ordering between the receptions of these two messages, as these reception events happen at different concurrent processes.

Another thing to be noted in the MSC of Figure 13 are the rectangles labeled s_0 and r_0, which annotate

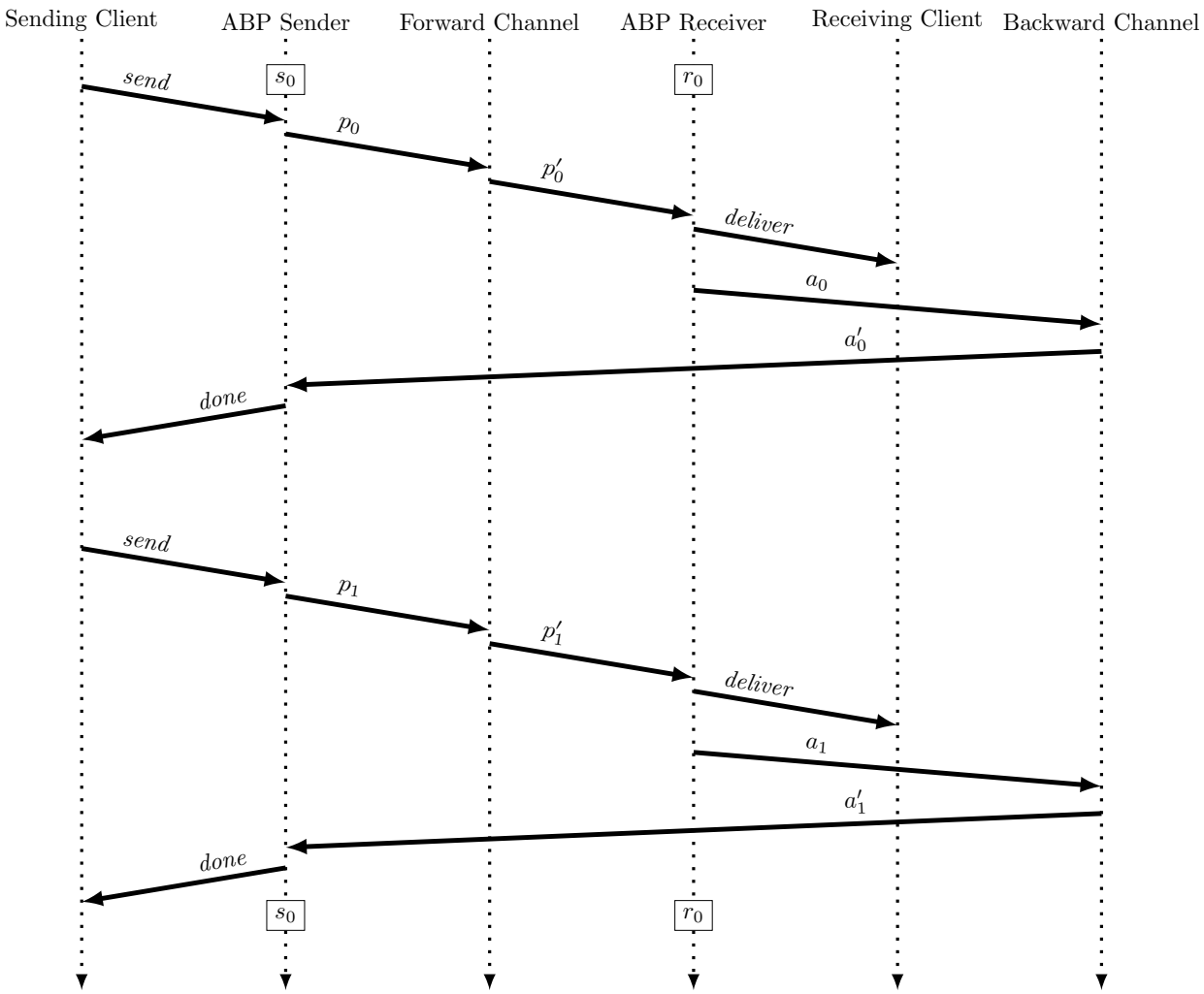


Figure 13: A scenario for the alternating-bit protocol.

the time-lines of *ABP Sender* and *ABP Receiver*. These correspond to states of these processes as will be explained next.

Starting from a scenario such as the one of Figure 13, incomplete processes can be constructed automatically for each of the processes participating in the scenario. For our purpose, which is synthesis, we do not need to construct incomplete processes for the environment components, since we already have complete processes for those (although we do need to check consistency between the scenario and the environment processes, in order to catch possible mistakes in any of these models). But we can use the scenario to automatically construct incomplete processes for the protocol components, in our case, *ABP Sender* and *ABP Receiver*.

The basic idea of the transformation of scenarios to incomplete processes is the following. For each component, we start at the top of each time-line, which is mapped by default to the initial state of the corresponding process, unless a state label indicates otherwise. Then we move down the time-line, and every time we encounter a new event, i.e., the transmission or reception of a message m, we create a new state and a transition from the last state to that state labeled by $m!$ or $m?$, depending on whether m is transmitted or received. An exception is when we encounter a state label in the time-line which has been encountered before. In that case, we do not create a new state, but direct the transition to the corresponding previously encountered state (this is the case, for instance, with state labels s_0 and r_0 on the time-lines of ABP Sender and Receiver processes in Figure 13).

Executing the above algorithm on the time-lines of *ABP Sender* and *ABP Receiver* of the scenario of Figure 13, results in the incomplete processes shown in Figures 14 and 15, respectively. The differences between these processes and the corresponding "manually" built sender and receiver of Figures 9 and 10 are explained in §5.3 that follows.

Note that the scenario of Figure 13 does not reveal all possible transitions in the system (let alone all possible behaviors, which are both infinite in number and infinite in length). But the scenario *does* cover all states of the protocol processes. This is important, since our completion method adds transitions, but not states. Therefore, this completion method works as long as the local states present in the input scenarios are sufficient (i.e., at least as many as a correct protocol requires for each process).

5.3 Automatic Completion of the Alternating-Bit Protocol

In view of automatic completion, we now revisit the ABP example presented in Section 4. The automatic completion algorithm described above has been implemented in a prototype tool written in Python. This tool is an extended version of the tool used in [6] (we thank Christos Stergiou for implementing these extensions). Using the completion tool, we synthesized the *ABP Sender* and *ABP Receiver* processes automatically, starting from incomplete versions of these processes. These incomplete versions were originally obtained from example scenarios as explained in Section 5.2. Other incomplete versions were obtained by further removing transitions from the original incomplete processes, in order to test scalability of the tool as described below.

The overall system architecture is as shown in Figure 1 and as described in Section 4.1. The environment processes given are the same as those depicted in Figures 2 and 3. The properties that the final system must satisfy, in addition to absence of deadlocks, are captured by the safety and liveness monitors of Figures 4, 5, 6, 7, and 8.

Incomplete Processes

The incomplete *ABP Sender* process provided as input to the automatic completion tool is shown in Figure 14. As can be seen from the figure, the incomplete sender is like the sender shown in Figure 9, but with several transitions missing (fair transitions are also not shown as these are automatically added during completion – see §3.3). In particular, all self-loop transitions are missing from states s_0 and s_4. Also, the self-loop transitions as well as the *timeout* transitions are missing from states s_2 and s_6. Note that the incomplete sender has no *timeout* transitions at all. In total, 12 out of 20 transitions are missing in the incomplete sender compared to the sender of Figure 9. As it is produced by the scenario of Figure 13, this incomplete

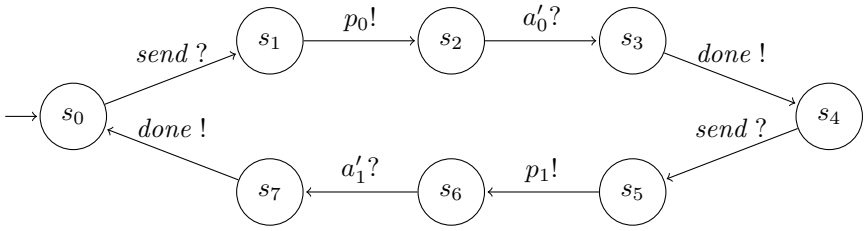


Figure 14: Incomplete process of ABP Sender.

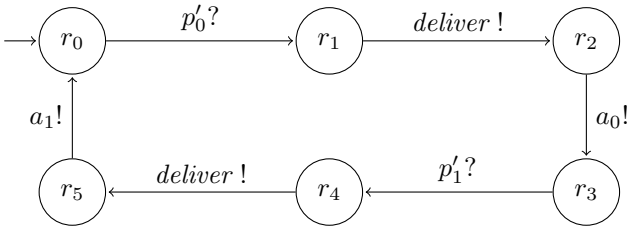


Figure 15: Incomplete process of ABP Receiver.

sender captures only the "typical" behavior of the protocol and does not specify how to deal with message loss and duplication. The logic to deal with these must be discovered by the completion tool.

The incomplete *ABP Receiver* process provided as input to the automatic completion tool is shown in Figure 15. It is like the receiver of Figure 10, but with 2 transitions missing, from states r_0 and r_3, respectively. Like the incomplete sender, the incomplete receiver only represents the typical behavior of the protocol.

Total Number of Completions in the ABP Example

Let us count the total number of completions in the case of the ABP example. First, let us count the number of possible completions of the incomplete ABP Receiver of Figure 15. To begin with, note that no transitions can be added in any of the states r_1, r_2, r_4, r_5. The reason is that these are output states and every completed process must be deterministic. There are two remaining states, r_0 and r_3. We count the number of transitions that may be added to each of them, and multiply the two numbers. First consider r_0. There are two input events for the receiver, p'_0 and p'_1. State r_0 already has a transition with p'_0: we cannot add an extra one with the same event, because that would break determinism. For event p'_1, we can add it on a transition leading to any of the 6 states of the receiver process; This makes 6 possible completions for state r_0. The same calculation holds for state r_3. Therefore, the total number of possible completions of the incomplete receiver is $6 \cdot 6 = 36$.

Let us now count the number of possible completions of the incomplete ABP Sender of Figure 14. States s_1, s_3, s_5, s_7 of the sender are output states, and thus cannot be completed due to determinism. There are 4 possible input events to the sender: *send*, *timeout*, a'_0, and a'_1. For each event missing from a state, there are 8 completions. Thus, there are $8 \cdot 8 \cdot 8 = 512$ possible completions for state s_0, since s_0 is missing three input events. The same calculation holds for states s_2, s_4, s_6. Therefore, the total number of completions of the sender is 512^4.

Multiplying the total number of completions of the sender with the total number of completions of the receiver, we get the total number of completions in the ABP example, which is $512^4 \cdot 36$, i.e., about 2.5 trillion completions. As mentioned above, each candidate completion needs to be model-checked for correctness. This example is sufficiently small for a model checker, so verification of each candidate does not take a lot of time. But even with 1 millisecond per candidate, it would take more than 78 years to verify all of them. As these numbers show, brute-force enumeration is not an option. Instead, the algorithm described in §5.1

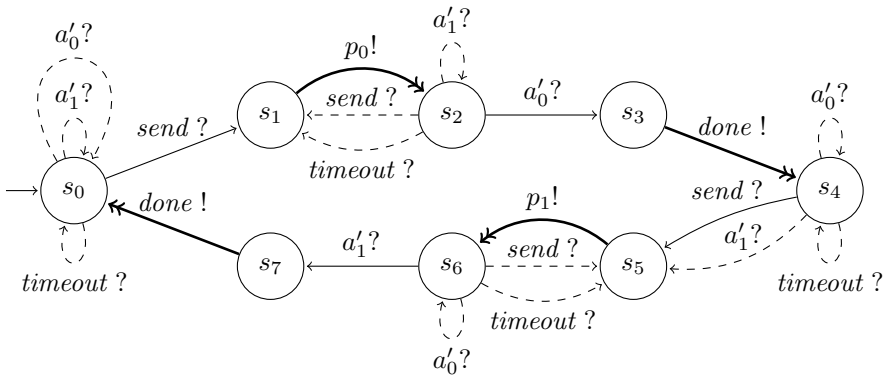


Figure 16: ABP Sender automatically synthesized by completing the sender of Figure 14.

is much more efficient, as we show next.

Automatic Completion of the ABP Example

We provide as input to our automatic synthesis tool: (1) the environment processes *Forward Channel*, *Backward Channel*, *Sending Client*, *Receiving Client*, and *Timer*, shown in Figures 2 and 3; (2) the incomplete sender and receiver processes shown in Figures 14 and 15; and (3) the safety and liveness monitors shown in Figures 4, 5, 6, 7, and 8. We ask the tool to solve the DPC problem (Problem 3), i.e., to synthesize deterministic sequential process completions for the ABP Sender and Receiver, such that the overall system is deadlock-free and satisfies the safety and liveness properties expressed by the monitors.

The tool runs for about 19 secs (on a T430s Lenovo laptop) and finds a solution consisting of 2 transitions added to the incomplete ABP Receiver and 12 transitions added to the incomplete ABP Sender. Completing the ABP Receiver with the 2 added transitions we find a process identical to the one of Figure 10. Completing the ABP Sender with the 12 added transitions we find the process shown in Figure 16. The added transitions are drawn with dashed arrows. As can be seen, the tool adds 3 transitions in each of the 4 input states of the sender process.

There are several similarities but also several differences between the automatically synthesized sender of Figure 16 and the manually built sender of Figure 9. On the similarities: state s_0 is completed in exactly the same way as in the manual design; also, the *timeout* transitions are added in exactly the same manner; finally, some (but not all) of the transitions labeled a_0', a_1' are added to states s_2, s_4, s_6 in exactly the same manner as in the manual design. But there are also several differences: the transitions labeled *send* are added differently at states s_2 and s_6; also the transition labeled a_1' is added differently at state s_4.

One might wonder whether some of these transitions (in particular those that differ in the automated and manual versions) really matter. For instance, are all the added transitions really necessary, or can some of them be removed without affecting the correctness of the protocol? Given our requirement that input states of completed protocol processes must be input-enabled, none of the added transitions can be completely removed. However, this does not imply that the exact placement of these transitions (i.e., their target states) matters. To check if it does, we use a feature of our tool which allows to identify *dead transitions*. These are local transitions (i.e., of some component process) which are never *exercised* in the global model, i.e., which never participate in any synchronized transition of the global state space.

We ask the tool to find dead transitions giving it as input the completed ABP model. The tool reports the following dead transitions (we only list dead transitions of the system processes, and not of the monitors):

- No dead transitions in Timer, Forward Channel, Backward Channel, ABP Receiver, and Receiving Client.
- Dead transition in Sending Client: $s_0^c \overset{done?}{\rightarrow} s_0^c$.

- Dead transitions in ABP Sender: $s_0 \overset{a'_0?}{\rightarrow} s_0$, $s_2 \overset{send?}{\rightarrow} s_1$, $s_4 \overset{a'_1?}{\rightarrow} s_5$, $s_6 \overset{send?}{\rightarrow} s_5$.

Armed with the above piece of information, we manually reset the 4 dead transitions of the ABP Sender to be self-loops in the incomplete sender process. Then we re-execute the synthesis tool, but this time asking it to find *all* possible solutions. After about 19 secs, the tool reports 4 solutions. In all 4 solutions the ABP Receiver is identical to the manually built receiver (Figure 10). All 4 solutions therefore correspond to 4 different variants of the ABP Sender. These variants are shown in Figure 17. They are identical, except for the transitions drawn with dashed lines.

As can be seen the 4 variants are the 4 possible combinations of placements of transitions labeled a'_1 and a'_0, respectively from states s_2 and s_6. These labels correspond to unexpected acknowledgments at those states. The "natural" solution is to ignore those acknowledgments, as is done with the self-loops of the fourth variant. However, it is also possible to return to the previous state (s_1 or s_5), as done in the other three variants. This then leads to retransmit the last message. It may be redundant and "wasteful" to retransmit when a wrong acknowledgment is received, but it is not incorrect.

As already stated, these 4 solutions were obtained by fixing the "don't care" dead transitions to be self-loops. These transitions can be set to lead anywhere, however, and we still get a valid solution. Since each of the 4 dead transitions may lead to any of the 8 possible target states, there are in total $4 \cdot 8^4 = 16384$ correct completions of the ABP example. It takes the tool about 88 mins to generate all 16384 solutions – see Table 1 and related discussion in the sequel.

Impossibility and Other Results

Apart from automatically completing the ABP example, we have also used the synthesis tool to obtain other interesting results, described next.

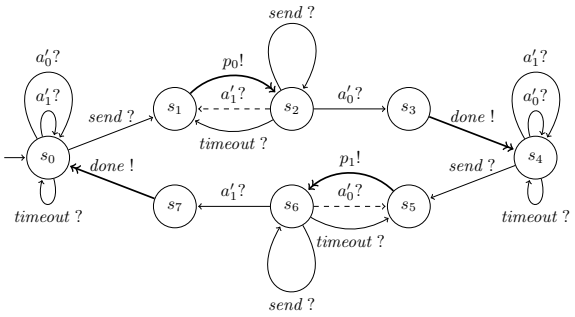
No solution exists if channels are unfair: We might wonder what happens if we remove the fairness constraints from the two channels. It takes about 18 secs for the tool to report that no correct completions can be found in that case. This impossibility result is to be expected since it is impossible to transmit reliably over an unreliable channel without assuming some kind of fairness from that channel. What is interesting here is that this impossibility result is obtained automatically.

No solution exists if *deliver* is not strongly fair: What if we remove the strong fairness assumption from the *deliver* transition of *ABP Receiver*? It takes the tool about 18 secs to report no solutions. This impossibility result is also to be expected, in view of the comments in §4.5.

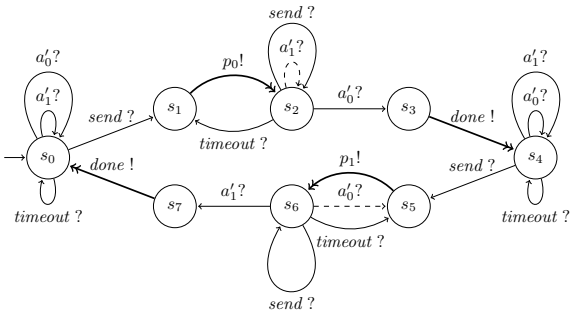
On the strong fairness of the *send* transition of *Sending Client*: As already stated in §4.5, the strong fairness condition on the *send* transition of *Sending Client* is needed only for the property of liveness monitor 3. What if we tried completion without this fairness assumption and without requiring this liveness property? Could this result in a different set of solutions? In fact, no. Performing this experiment we find that the completion tool returns the same 4 solutions as those shown in Figure 17.

A surprising ABP Receiver: One of the benefits of synthesis is that it offers even more surprises than verification. While performing one of the (too many to list exhaustively) experiments that we ran, we were surprised to see the tool return as a solution the ABP Receiver process of Figure 18. This process is identical to the one in Figure 10, except for the outgoing transition from state r_4. In Figure 18, this transition leads to state r_0 instead of r_5. (This solution was returned when starting with an incomplete receiver having fewer transitions than the one of Figure 15, in particular, missing the *deliver* transition from state r_4.)

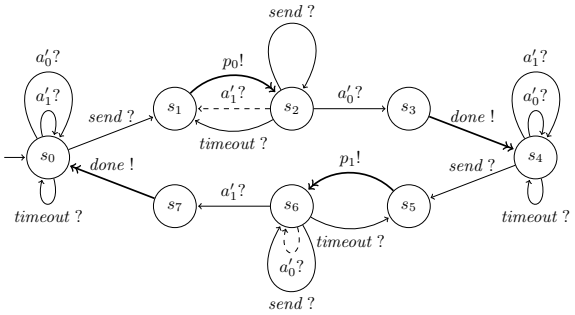
As it turns out, the receiver of Figure 18, together with any of the senders of Figure 17, is a correct solution to the ABP synthesis problem. This may appear surprising at first, as the receiver of Figure 18 omits to send the acknowledgment message a_1 after it has sent *deliver* from state r_4. But this omission simply results in the sender having to retransmit. Eventually, the receiver receives p'_1, moves from state



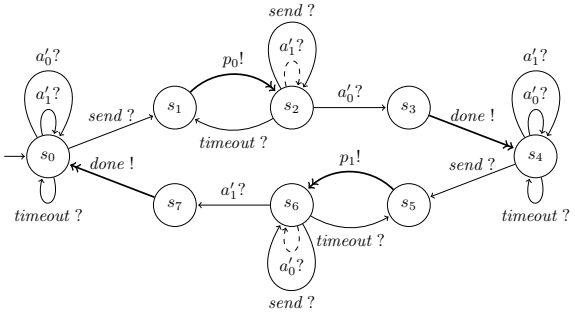
(a) Variant 1



(b) Variant 2



(c) Variant 3



(d) Variant 4

Figure 17: Four variants of ABP Sender automatically synthesized by the completion tool.

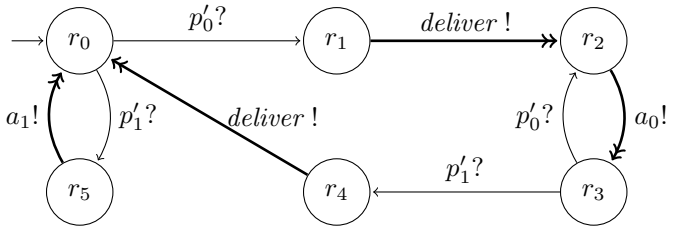


Figure 18: Automatically synthesized ABP Receiver during a completion experiment.

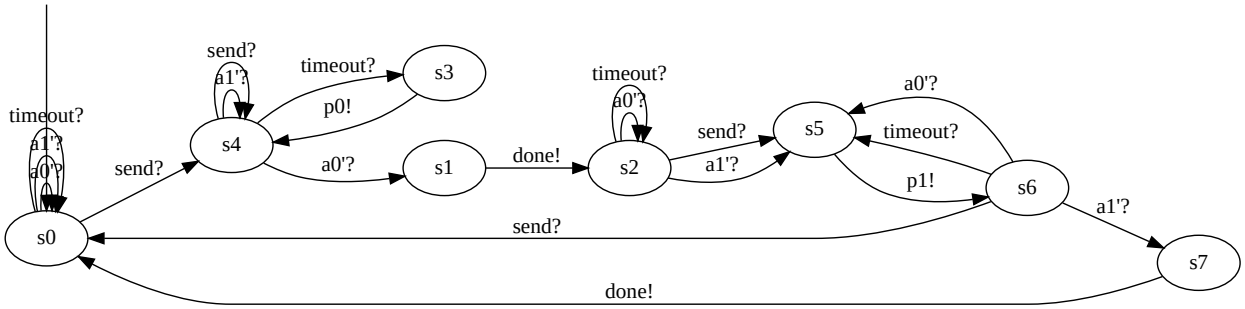


Figure 19: Automatically synthesized ABP Sender during a completion experiment.

r_0 to r_5, and sends the acknowledgment message a_1. As can be seen, this unorthodox solution is strictly speaking correct, even though it wastefully forces the sender to perform unnecessary retransmissions.

A surprising ABP Sender: Another solution that we found automatically and which surprised us at first was one where the ABP Sender was as shown in Figure 19 (this figure has been generated automatically by the tool using the Graphviz package – http://www.graphviz.org/). As can be seen in the figure, this sender does not immediately transmit p_0 after receiving *send* and moving to state s_4. Instead, it relies on *timeout* to move from s_4 to s_3, and then returns to s_4 having transmitted p_0. After that, its behavior is similar to the senders already presented earlier. Although unorthodox, this solution satisfies the requirements.

How many transitions can be left unspecified?

Ideally we should be able to synthesize the ABP protocol *from scratch*, by only specifying the number of states of the ABP Sender and Receiver processes, i.e., by providing incomplete processes with no transitions at all, and asking the completion tool to find the transitions. Our tool is not currently capable of that. However, the set of experiments that we report on in the sequel are encouraging, and lead us to believe that the goal of synthesizing ABP from scratch will soon be within reach.

We performed the following two sets of experiments:

1. Starting with the manually constructed ABP Sender of Figure 9, we removed transitions one by one, until we reached the incomplete sender of Figure 14. In each experiment we asked the completion tool to synthesize *all* possible correct completions. The results are shown in Table 1.

2. Starting with the incomplete sender of Figure 14, we continued removing transitions one by one, and in this case asking the tool to compute one solution each time. The results are shown in Table 2.

In all cases we used the incomplete ABP Receiver of Figure 15. In each experiment we ran the completion tool and measured its performance. Performance was measured in terms of execution time (on a T430s

Transitions removed from the complete sender of Figure 9	Solutions	Iterations	Time
$s_6 \xrightarrow{timeout?} s_5$	1	12	9 secs
$s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$	1	19	10 secs
$s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$	1	18	11 secs
$s_0 \xrightarrow{timeout?} s_0$, $s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$	1	25	13 secs
$s_0 \xrightarrow{timeout?} s_0$, $s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$, $s_0 \xrightarrow{a'_1?} s_0$	1	43	14 secs
$s_0 \xrightarrow{timeout?} s_0$, $s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$, $s_0 \xrightarrow{a'_1?} s_0$, $s_4 \xrightarrow{a'_0?} s_4$	1	56	16 secs
$s_0 \xrightarrow{timeout?} s_0$, $s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$, $s_0 \xrightarrow{a'_1?} s_0$, $s_4 \xrightarrow{a'_0?} s_4$, $s_2 \xrightarrow{a'_1?} s_2$	2	46	16 secs
$s_0 \xrightarrow{timeout?} s_0$, $s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$, $s_0 \xrightarrow{a'_1?} s_0$, $s_4 \xrightarrow{a'_0?} s_4$, $s_2 \xrightarrow{a'_1?} s_2$, $s_6 \xrightarrow{a'_0?} s_6$	4	72	19 secs
$s_0 \xrightarrow{timeout?} s_0$, $s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$, $s_0 \xrightarrow{a'_1?} s_0$, $s_4 \xrightarrow{a'_0?} s_4$, $s_2 \xrightarrow{a'_1?} s_2$, $s_6 \xrightarrow{a'_0?} s_6$, $s_0 \xrightarrow{a'_0?} s_0$	32	103	27 secs
$s_0 \xrightarrow{timeout?} s_0$, $s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$, $s_0 \xrightarrow{a'_1?} s_0$, $s_4 \xrightarrow{a'_0?} s_4$, $s_2 \xrightarrow{a'_1?} s_2$, $s_6 \xrightarrow{a'_0?} s_6$, $s_0 \xrightarrow{a'_0?} s_0$, $s_2 \xrightarrow{send?} s_2$	256	320	1 min 25 secs
$s_0 \xrightarrow{timeout?} s_0$, $s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$, $s_0 \xrightarrow{a'_1?} s_0$, $s_4 \xrightarrow{a'_0?} s_4$, $s_2 \xrightarrow{a'_1?} s_2$, $s_6 \xrightarrow{a'_0?} s_6$, $s_0 \xrightarrow{a'_0?} s_0$, $s_2 \xrightarrow{send?} s_2$, $s_4 \xrightarrow{a'_1?} s_4$	2048	2133	9 mins 20 secs
$s_0 \xrightarrow{timeout?} s_0$, $s_2 \xrightarrow{timeout?} s_1$, $s_6 \xrightarrow{timeout?} s_5$, $s_4 \xrightarrow{timeout?} s_4$, $s_0 \xrightarrow{a'_1?} s_0$, $s_4 \xrightarrow{a'_0?} s_4$, $s_2 \xrightarrow{a'_1?} s_2$, $s_6 \xrightarrow{a'_0?} s_6$, $s_0 \xrightarrow{a'_0?} s_0$, $s_2 \xrightarrow{send?} s_2$, $s_4 \xrightarrow{a'_1?} s_4$, $s_6 \xrightarrow{send?} s_6$	16384	16471	88 mins 19 secs

Table 1: Some performance experiments with our completion tool: in each experiment in this table all correct completions have been synthesized – their number is reported in the column "Solutions".

Lenovo laptop), and also in terms of number of *iterations*. Each iteration corresponds to one candidate completion tried out, i.e., it includes one call to the model checker to check correctness of that candidate.

As can be observed from Table 1, it is generally the case that the more transitions we remove, the more iterations (and time) it takes to find the solutions. However, this is not always the case, e.g., see the differences between rows 2 and 3, and also between rows 6 and 7. Also note that this table represents only one set of experiments corresponding to one of the many possible orders of removing transitions. Another order may yield different results, since not only the number of transitions removed but also the exact set of these transitions can influence performance significantly.

In Table 2 we start from the incomplete sender of Figure 14, which is already missing 12 transitions compared to the manual solution. We then further remove transitions as shown in the table. Note that removing these extra transitions results in some states of the sender process becoming deadlocks or even completely disconnected. These states can be completed to become either input states or output states. To help the tool, we specify the choice for each state as an input, i.e., we specify that states s_0, s_2, s_4, s_6 are input states, and that states s_1, s_3, s_5, s_7 are output states.

As with the experiments in Table 1, it is generally the case that the more transitions we remove the longer it takes to find a solution. But this is again not always true. For instance, removing the 2 transitions shown

Further transitions removed from the incomplete sender of Figure 14	Iterations	Time
None	65	19 secs
$s_0 \xrightarrow{send?} s_1$	168	24 secs
$s_0 \xrightarrow{send?} s_1,\ s_2 \xrightarrow{a'_0?} s_3$	1077	2 mins 4 secs
$s_0 \xrightarrow{send?} s_1,\ s_2 \xrightarrow{a'_0?} s_3,\ s_4 \xrightarrow{send?} s_5$	447	50 secs
$s_0 \xrightarrow{send?} s_1,\ s_2 \xrightarrow{a'_0?} s_3,\ s_4 \xrightarrow{send?} s_5,\ s_6 \xrightarrow{a'_1?} s_7$	4991	11 mins 36 secs
$s_0 \xrightarrow{send?} s_1,\ s_2 \xrightarrow{a'_0?} s_3,\ s_4 \xrightarrow{send?} s_5,\ s_6 \xrightarrow{a'_1?} s_7,\ s_1 \xrightarrow{p_0!} s_2$	5272	12 mins 22 secs
$s_0 \xrightarrow{send?} s_1,\ s_2 \xrightarrow{a'_0?} s_3,\ s_4 \xrightarrow{send?} s_5,\ s_6 \xrightarrow{a'_1?} s_7,\ s_1 \xrightarrow{p_0!} s_2,\ s_3 \xrightarrow{done!} s_4$	4056	9 mins 20 secs
$s_0 \xrightarrow{send?} s_1,\ s_2 \xrightarrow{a'_0?} s_3,\ s_4 \xrightarrow{send?} s_5,\ s_6 \xrightarrow{a'_1?} s_7,\ s_1 \xrightarrow{p_0!} s_2,\ s_3 \xrightarrow{done!} s_4,$ $s_5 \xrightarrow{p_1!} s_6$	9001	21 mins 44 secs
$s_0 \xrightarrow{send?} s_1,\ s_2 \xrightarrow{a'_0?} s_3,\ s_4 \xrightarrow{send?} s_5,\ s_6 \xrightarrow{a'_1?} s_7,\ s_1 \xrightarrow{p_0!} s_2,\ s_3 \xrightarrow{done!} s_4,$ $s_5 \xrightarrow{p_1!} s_6,\ s_7 \xrightarrow{done!} s_0$		aborted after 4 hours

Table 2: More performance experiments with our completion tool: in each experiment the first correct completion found has been returned.

in row 3 of the table requires more time (and iterations) to find a solution than removing the 3 transitions shown in row 4. But also note that the results of a single experiment can vary greatly, depending on the order in which transitions are explored in the algorithm. This order depends on a number of factors, such as on the solutions returned by the SAT solver (which is an external library). We can run the same experiment several times and obtain different results using the `-seed` option of the tool which controls the random seed. Using different random seeds yields, for example, a different solution for the experiment of row 2 is obtained in 475 iterations (instead of 168 shown in the table), and a different solution for the experiment of row 3 is obtained in 135 iterations (instead of 1077 shown in the table).

The last experiment shown in Table 2 did not terminate after 4 hours, and was then aborted. Note that in this last experiment the sender process provided to the tool has no transitions left, as all 8 transitions of the incomplete sender of Figure 14 have been removed. Being able to complete this last experiment would amount to synthesizing the ABP Sender from scratch, with the proviso that input-output states are also specified as an input.

5.4 Alternative Approaches

We describe two related approaches that can be used to solve the protocol completion problem.

Bounded Synthesis, Lazy Synthesis, and Template-Based Synthesis

Since distributed protocol synthesis is undecidable, the *bounded synthesis* problem asks, given a number k, do there exist protocol processes, each as a finite-state machine with at most k states, that satisfy the requirements of the desired solution [21]. The set of possible protocol processes is now bounded and thus bounded synthesis is decidable. The solution strategy in [21] is purely symbolic and can be used to reduce the distributed protocol completion problem that we have defined to Boolean satisfiability. The first step is a straightforward encoding of the desired transition functions of the protocol processes as Boolean variables. The problem of checking whether the product of the resulting processes satisfies all the safety and liveness monitors is then reduced to finding a satisfying assignment for a set of constraints over these variables. This relies on interpreting the monitors as *universal co-Büchi* automata and computing a bound on the lengths of runs that need to be explored to check acceptance.

Bounded synthesis is combined with *lazy synthesis* in [19]. Lazy synthesis uses a *solve-check-refine* loop which is similar to the approach described in §5.1. The main difference in our approach is the use of scenarios which are pre-processed into incomplete protocols. In contrast, the lazy synthesis algorithm in [19] works solely on the basis of an LTL specification and a bound on the number of states k, while it also contains an outer loop which increases k until a solution is found.

Completion of incomplete protocols is also similar to *template-based synthesis* [20]. The main difference is that in protocol completion transitions are added, whereas in template-based synthesis transitions are removed during the synthesis process.

Genetic Programming

An interesting alternative to distributed protocol synthesis relies on genetic programming [28, 29]: given an initial protocol template specified in a protocol description language and correctness requirements, if the model checker finds that the protocol does not satisfy the requirements, the tool tries multiple mutations of the abstract syntax tree of the protocol description, ranks the resulting versions by estimating how close they are to satisfying the requirements using state-space analysis, and iterates by probabilistically selecting a variant with weights proportional to ranks. The success of such a technique crucially depends on finding the suitable ranking function. This technique has been used to generate multiple variants of shared memory mutual exclusion algorithms and the leader election protocol [28, 29]. We believe that this approach is complementary to the one we have described and combining the two is a promising direction for future research.

6 Conclusions

We have described how current tools for automated analysis and constraint solving can be effectively used to assist a programmer in the design of distributed protocols. The main insights from our experience are summarized below:

Benefits of Protocol Completion The goal of formal verification is to produce a mathematical proof that an implementation meets its correctness specification. Model checking realizes this goal only partially but effectively: it checks only the requirements expressible in temporal logic against a finite-state abstraction of the protocol, but is supported by algorithmic tools that produce counterexamples that are useful for finding bugs in real-world protocols. The relationship of protocol completion to protocol synthesis can be potentially analogous to the one of model checking to verification: while protocol completion does not fulfill the original synthesis vision of deriving protocols automatically from high-level specifications, it can be useful as a design tool by *automatically inferring* missing cases in a partially designed protocol. Our case study of the ABP protocol illustrates the potential as the completion tool automatically infers how to cope with message losses and message duplications and synthesizes variants of the protocol that also meet the requirements.

Synthesis as Integration Classical synthesis aims to raise the level of abstraction, say, from imperative code to declarative logic formulas, and the goal of the synthesizer is to derive the low-level implementation from the high-level specification. In the more modern view of synthesis, a programmer interacts with the synthesizer by expressing the desired functionality via different *synthesis artifacts*. Such artifacts can include programs (that may not yet be complete), declarative specifications of high-level requirements, positive and negative examples of desired behaviors, and optimization criteria for selecting among alternative implementations. This diversity is aimed at allowing a programmer to express her insights through a variety of modes, leading to a potentially more intuitive and less error-prone way of programming. The synthesis tool *integrates* these different views about the structure and functionality of the system into a unified, concrete implementation using computational techniques such as decision procedures for constraint-satisfaction problems, iterative schemes for abstraction and refinement, and data-driven learning. The protocol completion problem we have described is an instance of this modern view of synthesis: the requirements expressed by

the monitors, the protocol template, and the scenarios describing some of the transitions are all different artifacts that are to be integrated into a unified implementation by the synthesis tool.

Importance of Formal Models Dating back to models such as CSP [24] and CCS [37], there is a rich literature in concurrency theory focused on variations and properties of formal models for distributed protocols. Model checkers however typically do not emphasize the nuances of formal models, and are primarily focused on the state-transition system underlying a protocol. As we have explained, the modeling assumptions, for instance, regarding fairness and non-blocking outputs, are crucial for automated synthesis to yield meaningful solutions. Thus, automated synthesis requires an integration of ideas from automated tools and concurrency theory.

Learning and Verification The CEGIS-based solution to protocol completion consists of interacting *learning* and *verification* phases where the learner proposes a candidate completion and the verifier checks the proposed completion for correctness and returns counterexamples if the correctness check fails. This is an instance of *active learning*, and such an architecture that integrates learning techniques with verification technology can have many potential applications in improving programmer productivity.

Future Work

We conclude by discussing two directions for future research:

Protocols as Extended FSMs In this paper, we modeled each protocol process as a finite-state machine. A more practical approach is to model a process as an *extended* finite-state machine, that is, a state-machine with variables (such as queues and counters) that the transitions may test and update. Such a protocol is an infinite-state system, but an effective heuristic for verification is to bound the values of all state variables (for example, the size of each queue) and then to apply a model checker. Inferring missing transitions for automatic protocol completion in this setting requires a systematic generation of expressions for the guards and updates for the variables. We have some preliminary work on this problem with application to cache coherence protocols [55, 7] (see also the approach based on genetic programming [28, 29]), but new computational techniques will be needed for applicability of this approach.

Battling the Exponential Search As noted earlier, solving the protocol completion problem requires battling two nested exponential search spaces: the number of possible completions of the given template and exploration of the state-space of each completion for violation of correctness requirements. While we have many techniques available for battling the state-space explosion, less is known for searching efficiently through the space of possible completions. Possible search strategies include enumeration with pruning, symbolic encoding, and stochastic walk over the graph of all possible completions [3]. A potential catalyst for advancing the state-of-the-art in computational solvers for this problem is the standardization of the related problem of *Syntax-Guided Synthesis* with a repository of benchmarks, prototype solvers, and an annual competition of solvers (see `www.sygus.org`).

Acknowledgments

This work was partially supported by the US National Science Foundation (awards 1329759, 1138996, and 1139138). We would like to thank Jennifer Welch and Bernd Finkbeiner for their helpful comments. We would also like to thank Milo Martin, Mukund Raghothaman, Christos Stergiou, and Abhishek Udupa, our co-authors of earlier work [6] upon which this paper is based. Special thanks go to Christos Stergiou for his invaluable help with modeling the alternating bit protocol and using the distributed protocol completion tool, as well as for implementing additional functionality in that tool.

References

[1] ITU Telecommunication Standardization Sector: ITU-R recommendation Z.120, Message Sequence Charts (MSC '96), May 1996.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM TOPLAS*, 15(1):73–132, 1993.

[3] R. Alur, R. Bodík, G. Juniwal, M.M.K. Martin, M. Raghothaman, S.A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 1–17, 2013.

[4] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[6] R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa. Synthesizing finite-state protocols from scenarios and requirements. In *Haifa Verification Conference*, LNCS 8855, pages 75–91. Springer, 2014. Extended version at CORR, abs/1402.7150.

[7] R. Alur, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa. Automatic completion of distributed protocols with symmetry. In *27th International Conference on Computer Aided Verification (CAV)*, LNCS 9207, pages 395–412, 2015.

[8] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: a case study. In *Proc. DATE*, pages 1188–1193, 2007.

[9] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition, 2010.

[10] A. Church. Logic, arithmetics, and automata. In *Proc. Int. Congress of Mathematicians*, pages 23–35, 1963.

[11] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[12] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6:217–232, 1995.

[13] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[14] L. de Alfaro and T. Henzinger. Interface automata. In *Foundations of Software Engineering (FSE)*. ACM Press, 2001.

[15] L. de Moura and N. Bjørner. Satisfiability Modulo Theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[16] D.L. Dill. The Mur*phi* verification system. In *Computer Aided Verification, 8th International Conference (CAV)*, LNCS 1102, pages 390–393, 1996.

[17] R. Ehlers, S. Lafortune, S. Tripakis, and M. Vardi. Supervisory Control and Reactive Synthesis: A Comparative Introduction. *Discrete Event Dynamic Systems*, pages 1–52, 2016.

[18] B. Finkbeiner. Synthesis of reactive systems. In *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 72–98. IOS Press, 2016.

[19] B. Finkbeiner and S. Jacobs. Lazy synthesis. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 219–234. Springer, 2012.

[20] B. Finkbeiner and H.-J. Peter. Template-based controller synthesis for timed systems. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 392–406, 2012.

[21] B. Finkbeiner and S. Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.

[22] R. Gawlick, R. Segala, J. Sogaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. In *Automata, Languages, and Programming, Proceedings of the 21st ICALP*, LNCS 820, pages 166–177. Springer-Verlag, 1994.

[23] A. Groce, K. Havelund, G. J. Holzmann, R. Joshi, and R. Xu. Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning. *Ann. Math. Artif. Intell.*, 70(4):315–349, 2014.

[24] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[25] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[26] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[27] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[28] G. Katz and D.A. Peled. Model checking-based genetic programming with an application to mutual exclusion. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference*, LNCS 4963, pages 141–156, 2008.

[29] G. Katz and D.A. Peled. Synthesizing solutions to the leader election problem using model checking and genetic programming. In *Hardware and Software: Verification and Testing - 5th International Haifa Verification Conference*, pages 117–132, 2009.

[30] O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *Proc. FOCS*, pages 531–540, 2005.

[31] H. Lamouchi and J. Thistle. Effective control synthesis for DES under partial observations. In *39th IEEE Conference on Decision and Control*, pages 22–28, 2000.

[32] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[33] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[34] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[35] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[36] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[37] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[38] G.L. Peterson and J.H. Reif. Multiple-person alternation. In *20th IEEE Symp. Found. of Comp. Sci.*, 1979.

[39] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, 1989.

[40] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31th IEEE Symposium on Foundations of Computer Science*, pages 746–757, 1990.

[41] V. Preoteasa and S. Tripakis. Towards Compositional Feedback in Non-Deterministic and Non-Input-Receptive Systems. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016.

[42] A. Puri, S. Tripakis, and P. Varaiya. Problems and examples of decentralized observation and control for discrete event systems. In *Synthesis and Control of Discrete Event Systems*, pages 37–56. Springer, 2002.

[43] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1), January 1987.

[44] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, January 1989.

[45] K. Rudie and J.C. Willems. The computational complexity of decentralized discrete-event control problems. *IEEE Transactions on Automatic Control*, 40(7), 1995.

[46] K. Rudie and W. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37, 1992.

[47] S.A. Seshia. Combining induction, deduction, and structure for verification and synthesis. *Proceedings of the IEEE*, 103(11):2036–2051, 2015.

[48] A. Solar-Lezama. Program sketching. *Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[49] A. Solar-Lezama, R.M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *Proc. 2005 ACM Conference on Programming Language Design and Implementation*, pages 281–294, 2005.

[50] J.G. Thistle. Supervisory control of discrete event systems. *Mathl. Comput. Modelling*, 23(11/12):25–53, 1996.

[51] J.G. Thistle. Undecidability in decentralized supervision. *Systems & Control Letters*, 54(5):503–509, 2005.

[52] S. Tripakis. Undecidable Problems of Decentralized Observation and Control. In *40th IEEE Conference on Decision and Control (CDC'01)*, pages 4104–4109. IEEE Computer Society, December 2001.

[53] S. Tripakis. Undecidable Problems of Decentralized Observation and Control on Regular Languages. *Information Processing Letters*, 90(1):21–28, April 2004.

[54] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A theory of synchronous relational interfaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(4), July 2011.

[55] A. Udupa, A. Raghavan, J.V. Deshmukh, S. Mador-Haim, M.M.K. Martin, and R. Alur. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–296, 2013.

[56] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.