Synthesizing Finite-state Protocols from Scenarios and Requirements

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

[1] University of Pennsylvania
[2] University of California, Berkeley
[3] Aalto University

Abstract. Scenarios, or Message Sequence Charts, offer an intuitive way of describing the desired behaviors of a distributed protocol. In this paper we propose a new way of specifying and synthesizing finite-state protocols using scenarios: we show that it is possible to automatically derive a distributed implementation from a set of scenarios augmented with a set of safety and liveness requirements, provided the given scenarios adequately *cover* all the states of the desired implementation. We first derive incomplete state machines from the given scenarios, and then synthesis corresponds to completing the transition relation of individual processes so that the global product meets the specified requirements. This completion problem, in general, has the same complexity, PSPACE, as the verification problem, but unlike the verification problem, is still hard (NP-complete) even for a constant number of processes. We present an algorithm for solving the completion problem, based on counterexample-guided inductive synthesis. We evaluate the proposed methodology for protocol specification and the effectiveness of the synthesis algorithm using the classical alternating-bit protocol, the VI cache-coherence protocol, and a consensus protocol.

1 Introduction

In formal verification, a system model is checked against correctness requirements to find bugs. Sustained research in improving verification tools over the last few decades 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 processes. The advances in these analysis tools now offer an opportunity to develop new methodologies for system design that allow a programmer to specify a system in more intuitive ways. In this paper, we focus on distributed protocols: the multitude of behaviors arising due to asynchronous concurrency makes the design of such protocols difficult, and the benefits of using model checkers to debug such protocols have been clearly demonstrated.

This work was partially supported by the Academy of Finland and by the NSF via projects *COSMOI: Compositional System Modeling with Interfaces* and *ExCAPE: Expeditions in Computer Augmented Program Engineering*. This work was also partially supported by IBM and United Technologies Corporation (UTC) via the iCyPhy consortium.

Traditionally, a distributed protocol is described using communicating finite-state machines (FSMs). The goal of this paper is to develop a methodology aimed at simplifying the task of designing them.

An intuitive way of specifying the desired behaviors of a protocol is by *scenarios*, where each scenario describes an expected sequence of message exchanges among participating processes. Such scenarios are used in textbooks and classrooms to describe the protocol and can be specified using the intuitive visual notation of Message Sequence Charts. In fact, the MSC notation is standardized by IEEE [1], and it is supported by some system development environments as design supplements. These observations raise the question: is it plausible to ask the designer to provide enough scenarios so that the protocol implementation can be automatically synthesized? Although one cannot expect a designer to provide scenarios that include all the possible behaviors, our key observation is that even a *representative* set of scenarios *covers* all the states of the desired implementation. The (local) states of a process are obtained from a scenario — using the explicit state-labels that appear as annotations as well as from the histories of events in which the process participates. If we consider all the states and the input/output transitions out of these states for a given process that appear in the given set of scenarios, we obtain a *skeleton* of the desired FSM implementation of that process. The synthesis problem now corresponds to *completing* this skeleton by adding transitions. This requires the synthesizer to infer, for instance, how to respond to a particular input event in a particular state even when this information is missing from the specified scenarios. The more such completions that the synthesizer can learn successfully, the lower the burden on the designer to specify details of each and every case. To rule out incorrect completions, we ask the designer to provide a model of the environment and correctness requirements. Some requirements such as absence of deadlocks can be generic to all the protocols, whereas other requirements can be specific to the coordination problem being solved by the protocol and given as finite-state monitors for safety and liveness properties in the form commonly used in model checkers. Note that scenarios and correctness requirements are used as under- and over-approximations of the behaviors of the protocol, respectively.

The synthesis problem then maps to the following *protocol completion* problem: given (1) a set of FSMs with incomplete transition functions, (2) a model of the environment, and (3) a set of safety/liveness requirements, find a completion of the FSMs such that the composition satisfies all the requirements. We show this problem, similar to the model checking problem, to be PSPACE-complete, but, unlike the model checking problem, to be NP-hard even for just one process. We present an algorithm for solving the protocol completion problem. The algorithm is an example of counterexample-guided synthesis [2]: candidates from the search space of completions are evaluated with respect to requirements and violations of the correctness requirements are used to prune the space.

To evaluate our methodology, we first consider the Alternating Bit Protocol (ABP), a classical solution to provide reliable transmission over unreliable channels. The canonical description of the protocol uses four scenarios to explain its behavior [3]. It turns out that the first scenario corresponding to the typical

behavior contains a representative of each local state of both the sender and receiver processes. Our algorithm for protocol completion is able to find a correct implementation from just one scenario, and thus, automatically learn how to cope with message losses and message duplications. We vary the input, both in terms of the set of scenarios and the set of correctness requirements, and study how it affects the computational requirements and the ability to learn the correct protocol. We also evaluate the effectiveness of scenarios on two other protocols: a cache coherence protocol and a distributed consensus protocol. In both cases, as in ABP, the scenarios produce automata that cover all the states of a desired implementation and our algorithm is able to synthesize the missing behaviors in a reasonable amount of time.

Related Work

Our work builds on techniques and tools for model checking [4] and also on the rich literature for formal modeling and verification of distributed protocols [5].

The problem of deriving finite-state implementations from formal requirements specified, for instance, in temporal logic, is called *reactive synthesis*, and has been studied extensively [6–8]. When the implementation is required to be distributed, the problem is known to be undecidable [9–12]. In *bounded synthesis*, one fixes a bound on the number of states of the implementation, and this allows algorithmic solutions to distributed synthesis [13]. Another approach uses *genetic programming* combined with model checking, to search through protocol implementations to find a correct one, which has been shown to be effective in synthesizing protocols such as leader election [14, 15].

Specifying a reactive system using example scenarios has also a long tradition. In particular, the problem of deriving an implementation that exhibits at least the behaviors specified by a given set of scenarios is well-studied (see, for instance, [16–18]). A particularly well-developed approach is *behavioral programming* [19] that builds on the work on an extension of message sequence charts, called *live sequence charts* [20], and has been shown to be effective for specifying the behavior of a single controller reacting with its environment. The work in [21] generalizes Angluin's learning algorithm to synthesize automata from MSCs but does not allow for the specification of requirements and relies on the programmer to answer classification and equivalence queries. More recently, scenarios — in the form of "flows" — have been used in the modular verification of cache coherence protocols [22].

Our approach of using both the scenarios and the requirements in an integrated manner and using scenarios to derive incomplete state machines offers a conceptually new methodology compared to the existing work. We are inspired by recent work on program sketching [23, 2] and on protocol specification [24]. PSKETCH [2] uses similar techniques but targets concurrent data structures and is limited to safety properties. Compared to TRANSIT [24] in this paper we limit ourselves to finite-state protocols but consider both safety and liveness requirements and provide a fully automatic synthesis procedure.

The protocol completion problem itself has conceptual similarities to problems such as *program repair* studied in the literature [25], but differs in technical details.

2 Methodology

We explain our methodology by illustrating it on an example, the well-known Alternating Bit Protocol (ABP). The ABP protocol ensures reliable message transmission over unreliable channels which can duplicate or lose messages. As input to the synthesis tool the user provides the following:

- The *protocol skeleton*: this is a set of processes which are to be synthesized, and for each process, the *interface* of that process, i.e., its inputs and outputs.
- The *environment*: this is a set of processes which are known and fixed, that is, are not to be synthesized nor modified in any way by the synthesizer. The environment processes interact with the protocol processes and the product of all these processes forms a *closed* system, which can be model-checked against a formal specification.
- A *specification*: this is a set of formal requirements. These can be expressed in different ways, e.g., as temporal logic formulas, safety or liveness (e.g., Büchi) monitors, or "hardwired" properties such as absence of deadlock.
- A set of *scenarios*: these are example behaviors of the system. In our framework, a scenario is a type of *message sequence chart* (MSC).

In the case of the ABP example, the above inputs are as follows. The overall system is shown in Figure 1. The protocol skeleton consists of the two unknown processes *ABP Sender* and *ABP Receiver*. Their interfaces are shown in the figure, e.g., ABP Sender has inputs a'_0, a'_1, and *timeout* and outputs *send*, p_0, and p_1. The environment processes are: *Forward Channel* (FC) (from ABP Sender to ABP Receiver, duplicating and lossy), *Backward Channel* (BC) (from ABP Receiver to ABP Sender, also duplicating and lossy), *Timer* (sends *timeout* messages to ABP Sender), *Safety Monitor*, and a set of *Liveness Monitors*.

As specification for ABP we will use the following requirements: (1) deadlock-freedom, i.e., absence of reachable global deadlock states (in the product system) (2) safety, captured using safety monitors, which guarantee that send and deliver messages alternate (3) Büchi liveness monitors, which accept incorrect infinite executions in which either a send message is not followed by a deliver, a deliver is not followed by a send, or a send never appears, provided that the channels are fair and that the processes do not indefinitely ignore input messages.

We will use the four message sequence charts shown in Figure 2 to describe the behavior of the ABP protocol. They come from a textbook on computer networking [3]. The first scenario describes the behavior of the protocol when no packets or acknowledgments are lost or duplicated. The second and the third scenarios correspond to the expected behaviors of the protocol in the event of the loss of a packet and in the event of the loss of an acknowledgment respectively. Finally, the fourth scenario describes the behavior of ABP on premature timeouts and/or packet duplication.

A candidate solution to the ABP synthesis problem is a pair of processes, one for the ABP Sender and one for the ABP Receiver. Such a candidate is a valid solution if: (a) the two processes respect their I/O interface and satisfy some additional requirements such as determinism (these are defined formally in

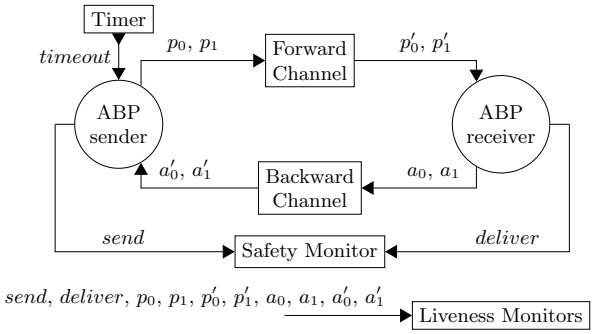

Fig. 1. ABP system architecture

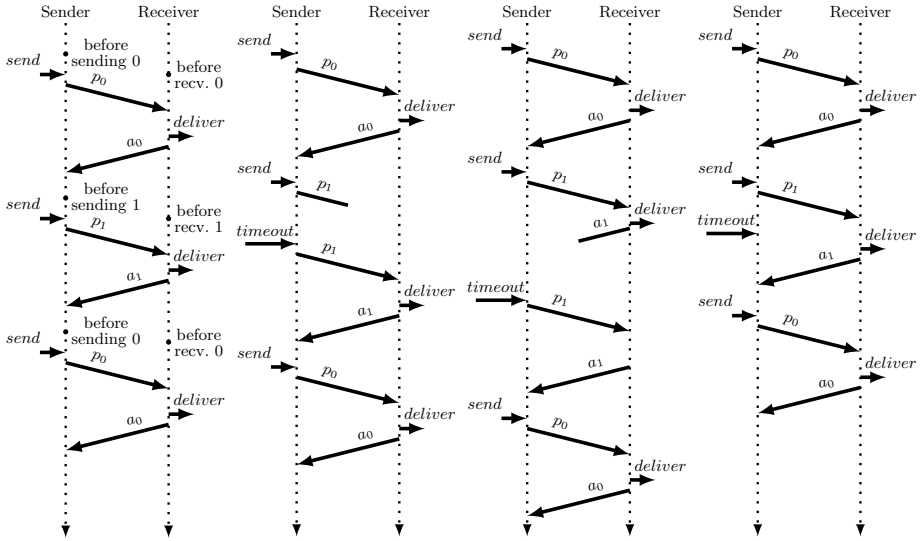

Fig. 2. Four scenarios for the alternating-bit protocol. From left to right: No loss, Lost packet, Lost ACK, Premature timeout/duplication.

Section 3.1), (b) the overall ABP system (product of all processes) may exhibit each of the input scenarios, and (c) it satisfies all correctness requirements.

Figure 3 shows for the ABP sender automaton, on the left, a manually constructed solution, and on the right, the output of the synthesis algorithm, when invoked with the requirements mentioned above and only the first scenario from Figure 2. It can be checked that the two instances of the ABP sender automaton are "similar" in the sense that they satisfy the same intuitive properties that one expects from the ABP protocol. In particular, the computed solution differs from the manual one in that it eagerly re-transmits p_0 when an unexpected acknowledgment a'_1 is received. This might incur additional traffic but satisfies all the safety and liveness properties for the ABP protocol. The computed solution for the ABP Receiver is the same as the manually constructed automaton.

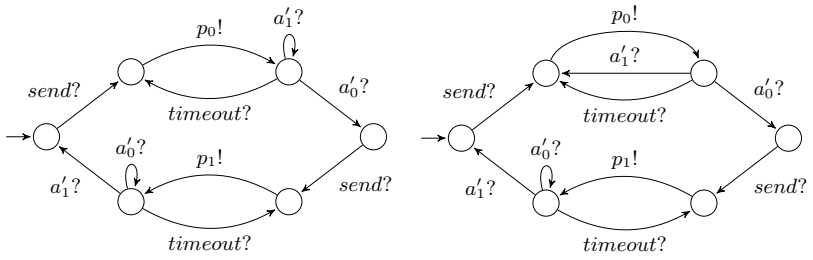

Fig. 3. ABP Sender "manual" solution (left) and solution computed by the synthesis algorithm using only the first scenario (right).

3 The Automata Completion Problem

We now describe how the problem described in Section 2 can be viewed as a problem of completing the transition relations of finite IO automata.

3.1 Finite-state Input-Output Automata

A *finite-state input-output automaton* is a tuple $A = (Q, q_0, I, O, T, O_f)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, I is a finite (possibly empty) set of inputs, O is a finite (possibly empty) set of outputs, with $I \cap O = \emptyset$, $T \subseteq Q \times (I \cup O) \times Q$ is a finite set of transitions,[4] and $O_f \subseteq O$ is a (possibly empty) set of outputs representing a fairness constraint.

We write a transition $(q, x, q') \in T$ as $q \xrightarrow{x?} q'$ when $x \in I$, and as $q \xrightarrow{x!} q'$ when $x \in O$. We write $q \to q'$ if there exists x such that $(q, x, q') \in T$. A transition labeled with $x \in I$ (respectively, $x \in O$) is called an *input transition* (respectively, an *output transition*).

A state $q \in Q$ is called a *deadlock* if it has no outgoing transitions. q is called an *input state* if it has at least one outgoing transition, and all outgoing transitions from q are input transitions. q is called an *output state* if it has a single outgoing transition, which is an output transition.

Automaton A is called *deterministic* if for every state $q \in Q$, if there are multiple outgoing transitions from q, then all these transitions must be labeled with distinct inputs. Determinism implies that every state $q \in Q$ is a deadlock, an input state, or an output state. Automaton A is called *closed* if $I = \emptyset$.

A *safety monitor* is an automaton equipped with a set of *error states* Q_e, $A = (Q, q_0, I, O, T, O_f, Q_e)$. A *liveness monitor* is an automaton equipped with a set of *accepting states* Q_a, $A = (Q, q_0, I, O, T, O_f, Q_a)$. A monitor could be both safety and liveness, in which case it is a tuple $A = (Q, q_0, I, O, T, O_f, Q_e, Q_a)$.

A *run* of an automaton A is a finite or infinite sequence of transitions starting from the initial state: $q_0 \to q_1 \to q_2 \to \cdots$. A state q is called *reachable* if there exists a finite run reaching that state: $q_0 \to q_1 \to \cdots \to q$. A safety monitor is called *safe* if it has no reachable error states. An infinite run of a liveness monitor

[4] The framework and synthesis algorithms can easily be extended to handle internal transitions as well, but we suppress this detail for simplicity of presentation.

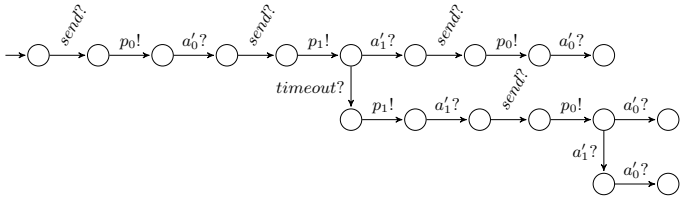

Fig. 4. Incomplete protocol automaton for ABP Sender using all scenarios from Figure 2, without using symmetric scenarios or labels.

is called *accepting* if it visits accepting states infinitely often. An infinite run is called *fair*, if for every $o \in O_f$, if it infinitely often visits some state q such that $o \in \{x \mid (q, x, q') \in T\}$ (o is "enabled" at q), then it makes a transition with output o infinitely often.[5] A liveness monitor is called *empty* if it has no infinite accepting fair runs.

3.2 Composition

We define an asynchronous (interleaving-based) parallel composition operator with rendezvous synchronization. Given two automata $A_1 = (Q_1, q_{0,1}, I_1, O_1, T_1, O_{f,1})$ and $A_2 = (Q_2, q_{0,2}, I_2, O_2, T_2, O_{f,2})$, the composition of A_1 and A_2, denoted $A_1 \| A_2$, is defined, provided $O_1 \cap O_2 = \emptyset$, as the automaton

$$A_1 \| A_2 \mathrel{\widehat{=}} (Q_1 \times Q_2, (q_{0,1}, q_{0,2}), (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, T, O_{f,1} \cup O_{f,2})$$

where $((q_1, q_2), x, (q_1', q_2')) \in T$ iff one of the following holds:

- $x \in O_1$ and $q_1 \xrightarrow{x!} q_1'$ and either $x \in I_2$ and $q_2 \xrightarrow{x?} q_2'$ or $x \notin I_2$ and $q_2' = q_2$.
- $x \in O_2$ and $q_2 \xrightarrow{x!} q_2'$ and either $x \in I_1$ and $q_1 \xrightarrow{x?} q_1'$ or $x \notin I_1$ and $q_1' = q_1$.
- $x \in (I_1 \cup I_2) \setminus (O_1 \cup O_2)$ and at least one of the following holds: (1) $x \in I_1 \setminus I_2$ and $q_1 \xrightarrow{x?} q_1'$ and $q_2' = q_2$, (2) $x \in I_2 \setminus I_1$ and $q_2 \xrightarrow{x?} q_2'$ and $q_1' = q_1$, (3) $x \in I_1 \cap I_2$ and $q_1 \xrightarrow{x?} q_1'$ and $q_2 \xrightarrow{x?} q_2'$.

During composition, the product automaton $A_1 \| A_2$ "inherits" the safety and liveness properties of each of its components. Specifically, a product state (q_1, q_2) is an error state if either q_1 or q_2 are error states. A product state (q_1, q_2) is an accepting state if either q_1 or q_2 is an accepting state.

Note that $\|$ is commutative and associative. So we can write $A_1 \| A_2 \| \cdots \| A_n$ without parentheses, for a set of n automata.

3.3 From Scenarios to Incomplete Automata

The first step in our synthesis method is to automatically generate from the set of input scenarios an *incomplete automaton* for each protocol process. The second

[5] Of the many notions of fairness which are discussed in literature, we have chosen one notion of fairness that is adequate for the case studies in this paper. Our approach can be extended to more general forms of fairness assumptions.

step is then to complete these incomplete automata to derive a complete protocol. In the sections that follow, we formalize and study the automata completion problem. In this section, we illustrate the first step of going from scenarios to incomplete automata, by means of the ABP example.

The idea for transforming scenarios into incomplete automata is simple. First, for every "swim lane" in the message sequence chart corresponding to a given scenario, we identify the corresponding automaton in the overall system. For example, in each scenario shown in Figure 2, the left-most lane corresponds to ABP Sender and the right-most lane to ABP Receiver. These scenarios omit the environment processes for simplicity. In particular channel processes are omitted, however, we will use a primed version of a message when referencing it on the process that receives it.

Second, for every protocol process P, we generate an incomplete automaton A_P as follows. For every message *history* ρ (ρ is a finite sequence of messages received or sent by the process) specified in some scenario in the lane for P, we create a state s_ρ in A_P. If $\rho' = \rho \cdot x$ is an extension of history ρ by one message x, then there is a transition $s_\rho \xrightarrow{x} s_{\rho'}$ in A_P. At this point, we check that the inputs and outputs of A_P are included in the interface of P in the protocol skeleton and that A_P is deterministic. Applying this procedure to the scenarios in Figure 2, we obtain the incomplete automaton shown in Figure 4 for the ABP Sender.

Third, scenarios are annotated with labels. As shown in the first scenario of Figure 2, labels appear between messages on swim lanes. These are used to merge the states that correspond to message histories that are followed by the same label. Merging occurs for states of a single scenario as well as across multiple ones if the same label is used in different scenarios. If consistent labels are given to the initial and final positions in all swim lanes of the scenarios the resulting incomplete automata can be made cyclic. Furthermore, labels are essential for specifying recurring behaviors in scenarios and the structure of the incomplete automaton depends on the number and positions of labels used.

Finally, it is often the case that different behaviors of a system are equivalent up to simple replacement of messages. For example, all the ABP scenarios express valid behaviors if p_0 and a_0 messages are consistently replaced with p_1 and a_1 messages respectively and vice-versa. Thus, our framework allows for scenarios to be characterized as "symmetric".

We annotate the swim lanes of the ABP Sender scenarios of Figure 2 with "before sending 0" and "before sending 1" labels, and the swim lanes of the ABP Receiver with "before receiving 0" and "before receiving 1" labels. We also add the symmetric scenarios by switching 0 messages with 1 messages. The resulting incomplete automaton for ABP Sender is shown in Figure 5.

3.4 Automata Completion

Having transformed the input scenarios into incomplete automata, the next step is to *complete* those automata by adding the appropriate transitions, so as to synthesize a complete and correct protocol. In this section we formalize this completion problem. We define two versions of the problem: a special version

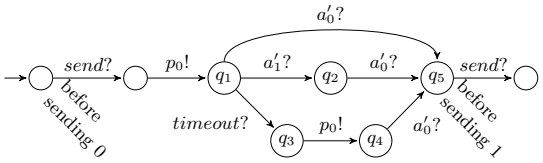

Fig. 5. Incomplete protocol automaton for ABP Sender from all scenarios of Figure 2 and their symmetric and after merging labeled states. (Only one half of the automaton is shown, the rest is the symmetric case for packet 1.)

with only a single incomplete automaton and a general version. In Section 4.1 we show that these problems are combinatorially hard.

Consider an automaton $A = (Q, q_0, I, O, T)$. Given a set of transitions $T' \subseteq Q \times (I \cup O) \times Q$, the *completion of A with T'* is the new automaton $A' = (Q, q_0, I, O, T \cup T')$.

Problem 1. Given automaton E (the *environment*) and deterministic automaton P (the *process*) such that $E \| P$ is defined, find a set of transitions T such that, if P' is the completion of P with T, then P' is deterministic and $E \| P'$ has no reachable deadlock states.

Note that if $E \| P$ is defined then $E \| P'$ is also defined, because, by definition, completion does not modify the interface (inputs and outputs) of an automaton.

Problem 2. Given a set of *environment automata* $E_1, ..., E_m$ (some of which can be safety or liveness monitors), and a set of deterministic *process automata* $P_1, ..., P_n$ such that $E_1 \| \cdots \| E_m \| P_1 \| \cdots \| P_n$ is defined, find sets of transitions $T_1, ..., T_n$ such that, if P_i' is the completion of P_i with T_i, then for $i = 1, ..., n$,

- P_i' is deterministic, for $i = 1, ..., n$,
- if the product automaton $\Pi := E_1 \| \cdots \| E_m \| P_1' \| \cdots \| P_n'$ is a safety automaton then it is safe,
- if Π is a liveness automaton then it is empty,
- and, Π has no reachable deadlock states.

4 Solving Automata Completion

In this section, we first show that Problems 1 and 2 are NP-complete and PSPACE-complete respectively. We then present a synthesis algorithm to solve the automata completion problem.

4.1 Complexity

It can be shown that Problem 2 is PSPACE-complete. Note that this is not surprising, as the verification problem itself is PSPACE-complete, for safety properties of distributed protocols. However, in the special case of one process and one environment automaton, while verification can be performed in polynomial time, a reduction from 3-SAT shows that the corresponding completion Problem 1 is NP-complete. The proofs are omitted due to lack of space, and can be found in [26].

Theorem 1. *Problem 1 is NP-complete and Problem 2 is PSPACE-complete.*

4.2 Synthesis Algorithm

We propose an algorithm for solving the automata completion problem that can be viewed as an instance of counter-example guided inductive synthesis [2]. At a high-level the algorithm works by maintaining a set of constraints on correct completions. The algorithm repeatedly chooses a candidate completion such that it satisfies these constraints. If this candidate completion satisfies the correctness requirements, the algorithm terminates. Otherwise, the information from the violation of the requirements is used to create more constraints on the set of correct completions and prune the search space.

We associate a Boolean variable with every candidate transition that can be added to the individual automata. The constraints maintained by the algorithm are propositional formulas over these transition variables. We initialize the constraint set with determinism and deadlock constraints. The first enforce that the protocol automata are deterministic, as described in subsection 3.1. For the second, we explore the reachable state space of the product of the environment and incomplete process automata; for every deadlock state, we add constraints that guarantee that at least one transition will be enabled out of that state. In the remainder of this section we will use t_i to refer both to transitions and their corresponding Boolean variables.

At the beginning of every iteration, a constraint solver — an ILP solver in our implementation — produces an assignment to the transition variables such the assignment satisfies the constraints. If the constraints are unsatisfiable, the algorithm concludes that no solution is possible and terminates. Otherwise, we translate the assignment to a set of transitions T, such that for every transition variable that the assignment sets to true, the corresponding transition is in T. Let $T = \{t_1, \ldots, t_n\}$. We complete the process automata with T, form their product with the environment automata, and monitors, and we check the absence of deadlocks, safety, and liveness violations using a model checker. The following cases are possible:

1. No violations are found. In this case, T is a correct completion, and the algorithm terminates.
2. A safety violation is found. This case means that the candidate solution T is incorrect. Moreover, any candidate T' obtained by adding extra transitions to T, i.e., $T' \supseteq T$, will also be incorrect, because adding extra local transitions can only add, but not remove, global transitions. This in turn implies that any reachable error state with T will also be a reachable error state with T', so any safety violation with T will also be a safety violation with T'. To enforce that no superset of T is included in any future candidate set, we add the formula $\neg(t_1 \wedge t_2 \wedge \ldots \wedge t_n)$ to the constraint set.
3. A liveness violation is found. This case also means that the candidate solution T is incorrect. A liveness violation, according to the definition of the problem 2, corresponds to a fair infinite accepting run, represented by a reachable cycle, that contains an accepting state of a liveness monitor. Although adding more transitions cannot eliminate the cycle, it is possible that additional transitions can render a fair run unfair: if a particular output $o \in O_f$

was not enabled in the cycle, then adding local transitions can cause o to become enabled. Let $T' = \{t'_1, \ldots, t'_m\}$ be the set of transitions that, if added, would make the infinite run unfair.[6] We add as a constraint the formula $\neg(t_1 \wedge t_2 \wedge \ldots \wedge t_n) \vee (t'_1 \vee t'_2 \vee \ldots \vee t'_m)$. The constraint guarantees that in all future candidate sets, the cycle will be unreachable, broken, or not fair.

4. A deadlock state is found. In this case, T is also incorrect, but could potentially be made correct by adding more transitions. Let $T' = \{t'_1, \ldots, t'_m\}$ be the set of candidate transitions such that, if any transition in T' is added, a transition is enabled out of the deadlock state. We add the constraint $(t_1 \wedge \ldots \wedge t_n) \rightarrow (t'_1 \vee \ldots \vee t'_m)$.

In every iteration, either a correct completion is found or the search space is pruned. We use an ILP solver to generate candidate sets from the constraints with an objective function that minimizes the size of the candidate set. In that way, in each iteration, we examine the smallest set of transitions that satisfies the constraints. This keeps the size of the product of the automata small and allows for faster checking of the properties.

We employ the following heuristic to prune the search space faster. Assume that a candidate set $T = \{t_1, \ldots, t_n\}$ is tested in an iteration of the algorithm and a safety violation is discovered. As described so far, the algorithm will remove all supersets of T from the search space by adding the constraint $\neg(t_1 \wedge \ldots \wedge t_n)$. However, if the safety violation is reachable by using only a subset of T, T'', then it is safe to also remove all supersets of T'' from the search space. Ideally, one would find all minimal subsets of T that alone can lead to a violation and remove all supersets of them. We approximate this by finding a minimal path to a safety violation using breadth-first search. If the path contains a subset of the transitions in T, we remove all supersets of that subset from the search space.

5 Evaluation

In this section we evaluate the effectiveness of scenarios and our methodology for specifying finite-state protocols. We use three benchmarks: the ABP protocol, a cache coherence protocol, and a consensus protocol. We first check whether the corresponding scenarios result in incomplete automata that cover all the states of a desired implementation. We then evaluate our synthesis algorithm on those benchmarks and investigate the effectiveness of scenarios in reducing the empirical complexity of the automata completion problem. Lastly, we discuss the interaction between the number of scenarios used to construct the initial incomplete automata and the number of requirements that are necessary to synthesize a correct protocol. A quantitative summary of our experiments can be found in Table 1. Each row corresponds to a combination of benchmark and set of input scenarios used for that benchmark, column "time" shows the total time that the synthesis algorithm took to find a correct completion, column "#

[6] For simplicity, we assume that process automata only communicate with environment automata. The constraint for the general case is more complicated but conceptually similar.

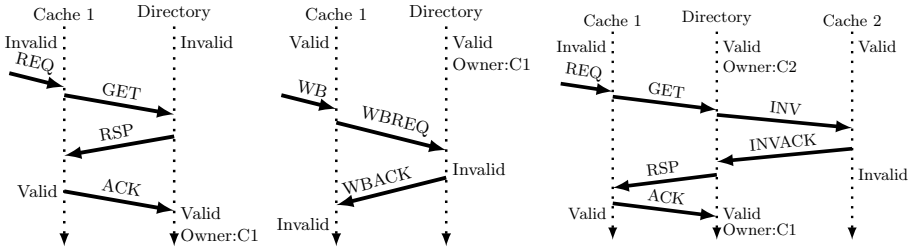

Fig. 6. Scenarios for the VI protocol

iterations" shows the number of iterations of the algorithm, i.e., the number of candidate sets of transitions tested, and "# candidate transitions" is the total number of candidate transitions for all process automata. Note that this last number, n, represents individual local transitions and not number of candidate completions. The size of the space of all possible completions is the number of subsets of the set of candidate transitions, i.e., 2^n.

5.1 Benchmarks

ABP This protocol was described in Section 2. We use different sets of input scenarios to create three versions of this benchmark. ABP1 used only the first scenario of Figure 2 to construct the incomplete automata, ABP2 used the second scenario, while ABP1-4 used all four scenarios.

We also construct a variation of the protocol that allows the clients to send different types of messages. In the protocol described in Section 2, only one type of message can be sent and received. In experiments ABPcolored1, ABPcolored2, and ABPcolored1-4, there are two types of messages that can be sent and received representing the different data that messages could carry.

VI protocol The VI protocol is a protocol for maintaining coherence among the private caches of a multi-processor system. The coherence requirement is that the value *read* by any processor is the same as the *last* value *written* to that location by any processor in the system. The scenarios shown in Figure 6 describe the working of the protocol. In the first scenario, Cache 1 acquires permissions to read or write to the cache block from the directory when no other processor in the system has permissions on the block. The second scenario demonstrates how a directory invalidates a cache that already has permissions on a block to fulfill the request of another cache for the block. These scenarios do not describe the behavior of the protocol when the second and third scenarios are interleaved, i.e., Cache 1 relinquishing permissions while Cache 2 attempts to acquire permissions.

We examine two variations of the VI protocol: one where there is a unique value for the data, in which case the protocol reduces to a distributed locking protocol (VI-no-data), and one where the data can take values 0 or 1, which captures the essence of the VI cache coherence protocol (VI).

Consensus In this problem we specify a protocol that describes how two processes can reach consensus on one value. Each process chooses initially a

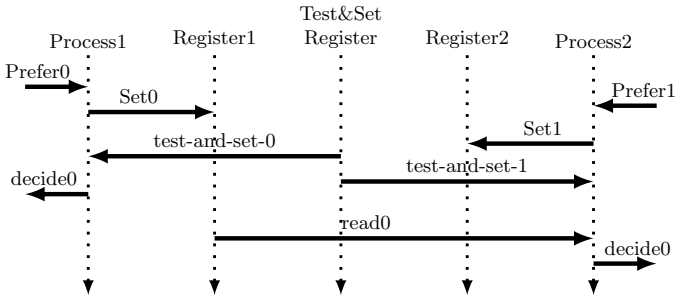

Fig. 7. Scenarios for the consensus protocol

preferred value and then they coordinate using shared memory to decide which of the two values to choose. The properties that the protocol has to satisfy are agreement (the two decisions must be the same), validity (the common decision must equal one of the preferred values), and wait-freedom (at any point, if only one process makes progress it will be able to make a decision). It has been shown that wait-freedom can be achieved only if a test-and-set register is used. The test-and-set register allows a process to write a value to it and read its previous value as an atomic operation

Figure 7 shows the scenario used for the consensus protocol. Both processes begin by non-deterministically choosing a value, messages "Prefer0" and "Prefer1", then write their choices in shared registers, "Register1" and "Register2", and then compete on setting the common test-and-set register which is initialized with 0. In this case, Process1 succeeds, the return value of the test-and-set operation is 0, and Process1 decides on its preferred value with message "decide0". On the other hand, Process2 fails, the test-and-set register returns 1, and Process2 reads the value chosen by Process1, and decides on that with messages "read0" and "decide0". We first attempt to synthesize the protocol starting from the incomplete automata constructed from the "success path", i.e., Process 1 lane of the scenario, and the "fail path", i.e., Process 2 lane. These two experiments correspond to rows "Consensus-success" and "Consensus-fail" of the Table 1. Finally, we implement a consensus protocol that does not use a test-and-set register, row "Consensus-no-test-and-set".

5.2 State coverage

We first observe that in all our experiments, except for "Consensus-success" and "Consensus-no-test-and-set", the states of the incomplete automata constructed by the scenarios cover all states of the protocols. In the "Consensus-success" experiment, the incomplete automaton is constructed using only the successful path of the protocol. A large part of the protocol's logic is missing from the input scenario, leaving the automaton with not enough states. The synthesis algorithm terminates and thus proves that no successful completion is possible. When we add an extra state in the incomplete automata without any edges to or from the rest of the states, the synthesis algorithm returns a completion that uses

Table 1. Quantitative summary of experiments.

Benchmark	time (s)	# iterations	# candidate transitions
ABP1	2.8	44	84
ABP2	9.9	87	172
ABP1-4	11.5	59	240
ABPcolored1	63.8	197	260
ABPcolored2	168.9	273	652
ABPcolored1-4	409.4	293	1012
VI-no-data	28.6	208	1170
VI	183.7	215	4538
Consensus-fail	0.3	5	264
Consensus-success	13.8	162	112
Consensus-success+1	21.4	163	216
Consensus-no-test-and-set	11.2	156	88

the extra state to implement the missing behavior. Row "Consensus-success+1" corresponds to that experiment.

5.3 Generalization and inference of unspecified behaviors

In all cases where the given scenarios covered all the states of the desired implementation the synthesis algorithm terminated with a correct completion. For the case of ABP with just one scenario specified, the algorithm successfully performs the generalization required to obtain a correct completion. The generalization performed is non-obvious: the correct protocol behaviors on packet loss, loss of acknowledgments and message duplication are inferred, even though the scenario does not describe what needs to happen in these situations. As can be seen in Figure 8, the incomplete automata constructed from the scenario describe only the protocol behavior over lossless channels. The algorithms are guided solely by the liveness and safety specifications to infer the correct behavior. In contrast, when all four scenarios are used, the scenarios already contain information about the behavior of the protocol when a single packet loss or a single message duplication occurs. The algorithm thus needs to only generalize this behavior to handle an arbitrary number of losses and duplications.

The same is true about the generalizations made by the algorithm in the other benchmarks. Specifically, in the case of VI, the synthesis algorithm correctly infers that in a complete protocol write-back and invalidate messages should be treated in the same way both from the caches and from the directory. Note that this behavior cannot be inferred by looking at caches and directory independently: they both have to implement it for the result to be correct.

5.4 Scalability

To validate our hypothesis that scenarios make the synthesis problem easier, we attempted to synthesize the ABP protocol with no scenarios specified, but with bounds on the number of states of the processes. These bounds were set to be equal to the corresponding number of states in the manually constructed version of the ABP protocol. We required that the protocol satisfy all the properties

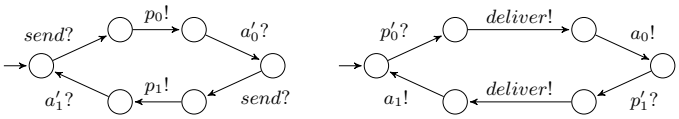

Fig. 8. Incomplete automata constructed from the first scenario of Figure 2.

discussed in Section 2. The synthesis algorithm ran out of time with no correct completion with a timeout of thirty minutes.

5.5 Scenarios and requirements

We observed that when fewer scenarios were used we needed to specify more properties — some of which were non-obvious — so that the algorithms could converge to a correct completion. For instance, when only one scenario was specified, we needed to include the liveness property that every deliver message was eventually followed by a send message. Owing to the structure of the incomplete automata, this property was not necessary to obtain a correct completion when all four scenarios were specified. Another property which was necessary to reject trivial completions when no scenarios were specified was that there has to be at least one send message in every run. Therefore, in some cases, using scenarios can compensate for the lack of detailed formal specifications.

6 Conclusions

The main contribution of this paper is a new methodology, supported by an automatic synthesis technique, for specifying finite-state distributed protocols using a mix of representative behaviors and correctness requirements. The synthesizer derives a skeleton of the state machine for each process using the states that appear in the scenarios and then finds a completion that satisfies the requirements. The promise of the proposed method is demonstrated by the ability of the synthesis algorithm to learn the correct ABP protocol from just a single scenario corresponding to the typical case. We would like to look at protocols that are best described using *extended* FSM with variables, such as more advanced cache-coherence protocols. In such cases, it will be necessary to synthesize symbolic guards and updates for each transition, see for example [24].

References

1. ITU Telecommunication Standardization Sector: ITU-R recommendation Z.120, Message Sequence Charts (MSC '96). (May 1996)
2. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '08
3. Kurose, J.F., Ross, K.W.: Computer Networking: A Top-Down Approach. 5th edn. Addison-Wesley Publishing Company, USA (2009)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2000)
5. Lynch, N.: Distributed algorithms. Morgan Kaufmann (1996)

6. Ramadge, P., Wonham, W.: The control of discrete event systems. IEEE Transactions on Control Theory **77** (1989) 81–98
7. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM Symposium on Principles of Programming Languages. (1989)
8. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3) (2012)
9. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: 31st Annual Symposium on Foundations of Computer Science. (1990) 746–757
10. Tripakis, S.: Undecidable Problems of Decentralized Observation and Control on Regular Languages. Information Processing Letters **90**(1) (April 2004) 21–28
11. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: IEEE Symposium on Logic in Computer Science. (2005) 321–330
12. Lamouchi, H., Thistle, J.: Effective control synthesis for DES under partial observations. In: 39th IEEE Conference on Decision and Control. (2000) 22–28
13. Finkbeiner, B., Schewe, S.: Bounded synthesis. Software Tools for Tchnology Transfer **15**(5-6) (2013) 519–539
14. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference. LNCS 4963 (2008) 141–156
15. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: Haifa Verification Conference. (2009) 117–132
16. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Transactions on Software Engineering **29**(7) (2003)
17. Uchitel, S., Kramer, J., Magee, J.: Synthesis of behavioral models from scenarios. IEEE Trans. Softw. Eng. **29**(2) (February 2003)
18. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (2012)
19. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM **55**(7) (2012) 90–100
20. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design **19**(1) (2001)
21. Bollig, B., Katoen, J., Kern, C., Leucker, M.: Learning Communicating Automata from MSCs. IEEE Transactions on Software Engineering **36**(3) (May 2010) 390–408
22. O'Leary, J., Talupur, M., Tuttle, M.R.: Protocol verification using flows: An industrial experience. In: Formal Methods in Computer-Aided Design, 2009. FMCAD 2009. (Nov 2009) 172–179
23. Solar-Lezama, A., Rabbah, R., Bodik, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation. (2005)
24. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: specifying protocols with concolic snippets. In: Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. PLDI (2013) 287–296
25. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Computer Aided Verification, 17th International Conference. LNCS 3576 (2005) 226–238
26. Alur, R., Martin, M.M.K., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Synthesizing finite-state protocols from scenarios and requirements. CoRR **abs/1402.7150** (2014)