# On-the-fly Synthesis of Scarcely Synchronizing Distributed Controllers from Scenario-Based Specifications

Christian Brenner[1], Joel Greenyer[2], and Wilhelm Schäfer[1]

[1] Software Engineering Group,
Heinz Nixdorf Institute
University of Paderborn, Germany
{cbr,wilhelm}@uni-paderborn.de
[2] Software Engineering Group
Leibniz Universität Hannover, Germany
greenyer@inf.uni-hannover.de

**Abstract.** Distributed systems consist of subsystems that usually need to coordinate with each other. Each subsystem must decide its actions locally, based on its limited knowledge. However, these decisions can be interdependent due to global requirements, i.e., one subsystem may need to know how another one decided. Complex communication can be required to exchange this knowledge. With rising complexity, a correct manual implementation of all subsystems becomes unlikely. Therefore, our goal is to automate the implementation process as far as possible. This paper presents a novel approach for the automatic synthesis of a distributed implementation from a global specification. In our approach, MSDs—a scenario-based specification language—can be used to intuitively, but formally define the requirements. The resulting implementation comprises one automaton for each subsystem, controlling its behavior. Contrary to similar approaches, we automatically add communication behavior to the system only when local knowledge is insufficient.

## 1 Introduction

Advanced driver-assistant systems with inter-vehicle communication or decentralized production systems are examples of software-intensive, distributed systems where multiple components interact with each other and the environment to fulfill complex, sometimes critical requirements. These requirements often span multiple components, which must synchronize so that each component has sufficient information about the overall system state in order to act accordingly. Architectural constraints may prohibit the direct communication between certain components or may require economical use of channels; also the time needed for exchanging additional synchronization messages can be an issue. Therefore, the naive approach of full synchronization among all components is usually not feasible. With rising complexity, implementing a global specification correctly and with feasible synchronization becomes an extremely difficult task.

In this paper, we propose a novel algorithm for synthesizing distributed controllers from a Modal Sequence Diagram (MSD) specification. MSDs [9] are a variant of Live Sequence Charts (LSCs) [4], which allow engineers to describe what the system components of a system may, must, or must not do.

Approaches for synthesizing distributed controllers from MSD/LSC specifications have been described previously. One part of these approaches considers the question whether a distributed implementation exists where the subsystems exchange messages exactly as defined in the specification, without adding extra messages for synchronization among the distributed subsystems. The engineers have to manually ensure that this property is fulfilled by explicitly specifying all necessary communication. This becomes harder with rising complexity, increasing the chance of errors. The other part of these approaches asks whether a distributed implementation of a specification exists where extra synchronization messages can be added. The existing approaches of this kind automatically add synchronization messages such that all subsystems have perfect information about the global state of the system—even when this information is not required to act correctly according to the specification. This causes a large overhead due to the unnecessary communication and removes all parallelism from the system. In real-time systems, this unnecessary communication can even lead to a violation of timing requirements. But even when timing is not an issue, the given architecture might not allow all subsystems to communicate.

Contrary to existing approaches, our automatic algorithm introduces synchronization messages *scarcely*, only when the subsystems could not otherwise avoid violating the specification. Moreover, these synchronization messages are only added where allowed by the given architecture. Our algorithm explores candidate implementations of an MSD specification *on-the-fly* and can often find a solution without constructing all alternatives. This is an advantage over related approaches that start with constructing a maximal global controller and then attempt to distribute it. A further advantage over most related approaches is that we consider specifications where also *assumptions* can be described on how the system's environment may, will, or will not behave.

This paper is structured as follows. Section 2 introduces the foundations and a running example. Section 3 introduces our distributed synthesis approach. We illustrate its application for the running example in Sect. 4. We present related work in Sect. 5 and conclude with Sect. 6.

## 2 Foundations

As an example, we consider a simple production system with one robot arm and a press (see sketch in Fig. 1). Blanks arrive on a feed belt where, at its end, a sensor detects the arrival of a blank and whether it is intact or broken. The arm must remove broken blanks and move intact ones to the press, where the blanks are pressed into plates. After pressing, the arm must transport the pressed plates to a deposit belt. We assume that until the arm has delivered the plate or removed a broken blank, no new blanks will arrive.

The system has two software controller components. One receives signals from the sensor and controls the robot arm, the other controller controls the press. Figure 1 shows the requirements (R1-R5) and assumptions (A1, A2).

### 2.1 MSD Specifications

MSDs are a variant of LSCs [4,10], proposed by Harel and Maoz as a formal interpretation of UML sequence diagrams [9]. MSDs specify the interaction behavior of components or *objects*. We consider open systems with controllable *system objects* and uncontrollable *environment objects*. Together, these objects form the *object system*. An MSD specification consists of an object system and a set of MSDs, which can be *requirement MSDs* and *assumption MSDs*.

**Lifelines, Messages, MSD Semantics** An MSD contains lifelines that each represents one object. Objects can exchange messages. A message has a name and one sending and one receiving object. We only consider *synchronous* messages, where the sending and receiving of a message together form a single *event*, also called *message event*. A *run* of a system is an infinite sequence of events.

We model the object system by a UML composite structure diagram (CSD, see Fig. 1). System objects have a rectangular shape; environment objects have a cloud-like shape. Connectors define which objects can exchange messages.

An MSD contains *(diagram) messages* that have a name and a sending and receiving lifeline. They also have a *temperature* and an *execution kind*, which indicate safety and liveness requirements, as we will explain shortly. The temperature can be either *hot* or *cold*. The execution kind can be either *executed* or *monitored*. In Fig. 1, the temperature and execution kind is annotated by labels (c,m), (c,e), (h,m), (h,e) next to the messages. In addition, the hot message arrows are colored red; cold message arrows are colored blue. Monitored messages have a dashed arrow; the arrows of executed messages have a solid line.

A message in an MSD can be *unified* with a message event if the sending and receiving object of the message event are represented by the sending and receiving lifeline of the diagram message. If a message event occurs that can be unified with the first message of an MSD (we assume that there is always exactly one first message), then an *active copy* of the MSD, also called *active MSD*, is created. The active MSD progresses as further events occur that can be unified with subsequent messages. This progress is indicated by the *cut*, which marks messages that have been unified. The MSD labeled R1 in Fig. 1 shows the cut as a dashed horizontal line spanning all lifelines (the cut is not part of the specification, but only part of its interpretation). The cut here indicates that the events `intactBlank` and `blankToPress` occurred. There can be several active MSDs at the same time. The occurrence of `blankToPress` for example also activated the MSDs labeled R2 and R4. If the cut reaches the end of the MSD, the active copy of the MSD is terminated and discarded.

A message in an active MSD is called *enabled* if the cut is immediately in front of the message on the sending and receiving lifeline. If a hot message is

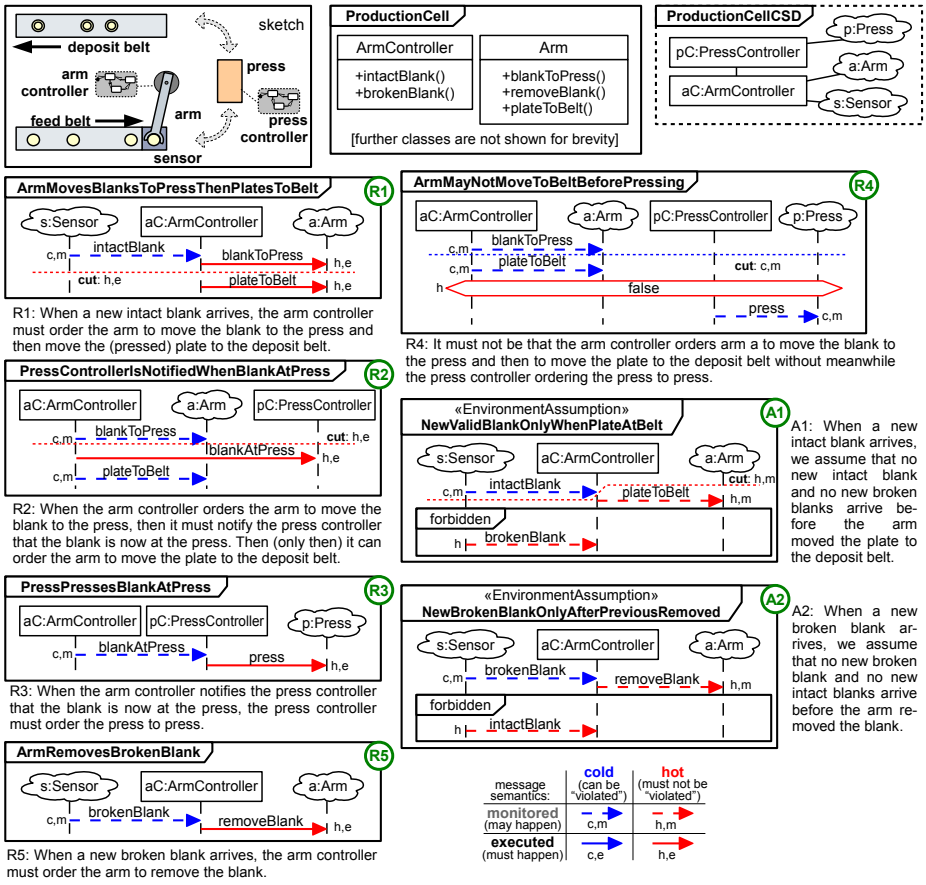


**Fig. 1.** The production cell MSD specification

enabled, the cut is also hot; otherwise the cut is cold. If an executed message is enabled, the cut is also executed; otherwise the cut is monitored. Labels also indicate the cut temperature and execution kind in Fig. 1.

If a message event occurs that can be unified with a message in the same MSD that is not currently enabled, this is called a *violation*. If the cut is hot, a violation must not happen. If it does, this is called a *safety violation*. If the cut is cold, a violation is allowed to happen and results in a premature termination of the active MSD (*cold violation*). If the cut is executed, this means that the cut must eventually progress, otherwise this is called a *liveness violation*.

An MSD can also contain *conditions*. In this paper, we only have, and thus only explain, the hot `false` condition in MSD R4, depicted by a red hexagon. In this case, the cut must not reach the condition, otherwise this is a safety violation. This MSD is an example of an *anti-scenario*, which describes a sequence of events that must not happen; in this case `blankToPress` followed by `plateToBelt`. The

only way this sequence is allowed to happen is if `press` occurs in between, which leads to a cold violation in the cut illustrated in Fig. 1.

**Assumption MSDs, Satisfying an MSD Specification** A run of a system is *accepted* by an MSD iff it does not lead to a safety or liveness violation of this MSD. The set of all runs accepted by an MSD $D$ is also called the *language* of this MSD, $L(D)$. Given a set of MSDs $M = \{D_1, \ldots, D_n\}$, the *language* of this set of MSDs, $L(M)$, is the set of runs accepted by all the MSDs in the set, $L(M) = \bigcap_{i=1}^{n} L(D_i)$.

We consider MSD specifications $MS$ that comprise two sets of MSDs, *requirement MSDs* $G$ (for "guarantees") and *assumption MSDs* $A$. The set of runs *satisfying* an MSD specification, $L(MS)$, is defined as $L(MS) = \overline{L(A)} \cup L(G)$, where $\overline{L(A)}$ is the set of all runs not in $L(A)$. Intuitively, a run satisfies an MSD specification iff it satisfies the requirements or does not satisfy the assumptions.

We consider open systems consisting of *environment objects* and *system objects*. Usually, we use assumption MSDs to constrain the possible behavior of the environment, hence we also call these MSDs *environment assumptions*. Assumption MSDs have the additional label «EnvironmentAssumption»; the MSDs A1 and A2 in Fig. 1 are examples for such assumption MSDs. Moreover, we assume, as also Harel et al. [10], that the system objects can send any finite number of messages between two messages sent by environment objects.

**The Specification State Graph** An MSD specification induces a transition system that we call the *specification state graph (SSG)*. This graph is the basis for our algorithm. The SSG consists of states and transitions; the transitions are labeled with message events and a state represents a set of active MSDs with a particular configuration of cuts. The start state is a state with no active MSDs. The cut configuration of the other states is the configuration that results after any sequence of message events that corresponds to a path in the SSG from the start state to that state. Transitions labeled with system events are *controllable*; those with environment events are *uncontrollable*. The SSG can be considered a *game graph*, representing a game played by the system against the environment.

The synthesis must create a strategy for choosing controllable transitions such that always eventually a *goal* state can be reached (Büchi condition). The primary form of goal state is a state without any enabled executed message in any active requirement MSD. In these states, intuitively, the system currently has no obligations to do anything and waits for the next environment event to happen. While calculating this strategy, the algorithm must assume that the environment can do anything to keep the system from reaching such a state. The strategy must avoid safety violations of requirement MSDs, unless they coincide with safety violations of assumption MSDs; this represents behavior that we assume is not possible to occur in the environment. Moreover, safety violations in requirement MSDs are allowed while executed messages are enabled in assumption MSDs; this represents environment behavior that we assume is not complete—maybe

here the environment only achieves a violation of the requirements at the expense of finally violating the assumptions, too. These latter states are also goal states.

More formally, a *strategy* is a subset of transitions of an SSG. A strategy is *winning* under the following two conditions. *W1*: The SSG, by taking only these strategy transitions, contains no deadlocks and no cycles without goal states. *W2*: If a strategy includes an outgoing uncontrollable transition of a state, it must include all outgoing uncontrollable transitions of that state. Intuitively, *W2* means that if the environment makes a move, the strategy must consider all its possible moves; *W1* means that the Büchi condition is satisfied.

### 2.2 The Controller System

Our synthesis approach generates a *controller system* consisting of *controllers* that together implement a given MSD specification. We consider controllers to be deterministic finite-state automata that define which messages are sent at what point in time for one object each. A set of controllers $CS$ is called a *controller system* if each system object is controlled by a controller.

We call an event *controllable by an object* if the object sends the event. We call it *observable by an object* if the object sends or receives the event. Events are controllable/observable for a controller if they are controllable/observable for the object it controls. We call transitions controllable/observable for an object or controller if their event is controllable/observable.

By parallel composition, a controller system can be mapped to an equivalent *global (controller) automaton*. Given an MSD specification for the same object system, the states of the global automaton correspond to states of the specification's SSG. We call an SSG state *reachable* for a controller system iff there exists a sequence of events for which there is a corresponding path in the global automaton as well as in the SSG (starting from their resp. start states). The transitions between the SSG states that are reachable for a controller system define a *strategy corresponding to the controller system*.

## 3 Algorithm for Distributed Synthesis

Our algorithm creates a controller system that implements a given MSD specification—provided that an implementation is possible. A controller system implements a specification iff it corresponds to a winning strategy (see Sect. 2.1) in the SSG. Such a controller system is a *solution* for the distributed synthesis.

Our algorithm systematically generates *candidate controller systems* (in short *candidates*), i.e., controller systems which might be extensible towards a solution. These candidates are extended in such a way that an increasing number of SSG states fulfill the winning conditions. The algorithm backtracks if it finds that a candidate cannot be extended into a solution. To allow for this systematic search, the candidates are maintained as nodes in a graph structure called the *candidate graph (CG)*. The algorithm performs a depth-first search (DFS) in the CG to approach a solution (cf. Fig. 2):

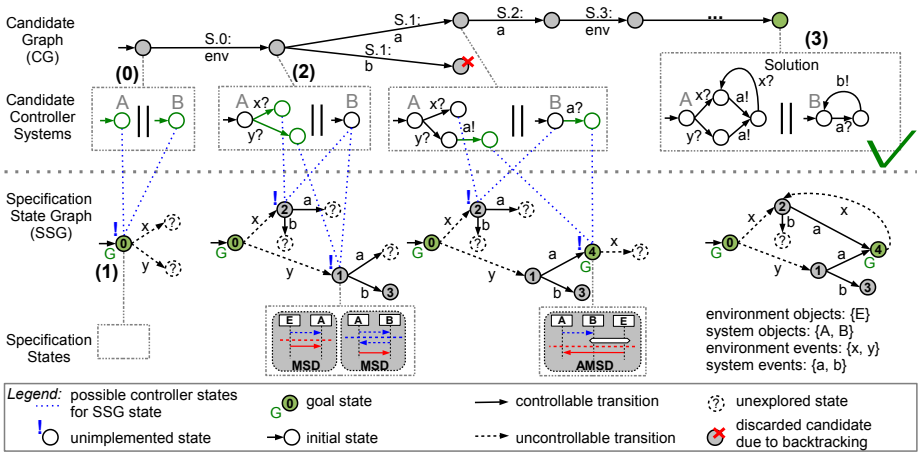


**Fig. 2.** Overview of the algorithm

The initial CG consists of a candidate with only the start state for each controller of a system object (0). We assume that environment objects can send any sequence of environment events and, hence, we do not need to construct controllers for them. In each step, the algorithm considers a *current candidate*. The algorithm finds all states in the SSG that are reachable for this current candidate. Then the algorithm checks whether any of these currently reachable SSG states violate the winning conditions (cf. Sect. 2.1) (1). We call these SSG states *unimplemented states*. If no such state exists, the candidate is a solution and the DFS terminates (3). Otherwise, the DFS extends the CG by constructing a successor candidate (2). The successor candidate is created by extending a copy of the current candidate and adding transitions and states to ensure that one of the previously unimplemented SSG states fulfills the winning conditions and becomes *implemented*. If this is possible, the algorithm performs the next step with the successor candidate. If an unimplemented state cannot be implemented, i.e., a candidate is not extensible towards a solution, the DFS backtracks.

We explain the two procedures of *computing unimplemented states* and *creating successor candidates* in more detail in Sect. 3.1 and 3.2.

### 3.1 Computation of Reachable Unimplemented SSG states

Identifying reachable unimplemented SSG states works as follows:

**Identify reachable SSG states**: We call a specification state $s$ and a controller state $c$ *corresponding* if there is a sequence of events after which the specification is in state $s$ and the controller is in state $c$. We call a controller transition $t_c$ and a specification transition $t_s$ *corresponding* if the source states are corresponding and both transitions are labeled with the same event. The algorithm computes these correspondences according to the following definition of the relation *corr* of corresponding states of an SSG $S$ for a controller $C$:

1. $(C.initial, S.initial) \in corr$.
2. For transition $s \xrightarrow{e} s'$ of $S$, $e$ unobservable for $C$:
   $(c, s) \in corr \Rightarrow (c, s') \in corr$.
3. For transition $s \xrightarrow{e} s'$ of $S$, $e$ observable for $C$ (it exists $c \xrightarrow{e} c'$ in $C$):
   $(c, s) \in corr \Rightarrow (c', s') \in corr$.

As we will discuss shortly in Sect. 3.2, the controllers may be extended with transitions that send or receive additional messages for synchronization of the controllers. These messages do not appear in the MSD specification, and thus there are no corresponding transitions in the SSG. For the sender of a synchronization message, the corresponding SSG states are the same for the source and target states of the transition labeled with the synchronization event. For the receiver of the synchronization message, however, the target state of the synchronization transition will only correspond to the SSG states that correspond to the source states of the synchronization transitions of both the sender and receiver. Intuitively, the sender of a synchronization message conveys its information about the possible global system states to the receiver. Conversely, the receiver does not convey its information about the global state to the sender, because we assume that the receiver cannot block the sending of messages.

The definition of the relation *corr* is extended as follows.

4. For synchronization sending transition $c \xrightarrow{synch!} c'$ in $C$:
   $(c, s) \in corr \Rightarrow (c', s) \in corr$.
5. For transition $c \xrightarrow{synch?} c'$ in $C$ that receives a synchronization message sent by transition $c2 \xrightarrow{synch!} c2'$ in another controller $C2$:
   $(c, s) \in corr \wedge (c2, s) \in corr \Rightarrow (c', s) \in corr$

**Determine whether reachable SSG states are implemented**: We call an event *e implemented in a reachable SSG state s* if the sending and receiving objects either are environment objects or their controllers define a sending/receiving transition for $e$ in *all* their controller states that are corresponding to $s$. Additionally, the outgoing transition for $e$ in the SSG state $s$ may close a loop of SSG transitions for other implemented events only if that loop includes a goal state. The latter is required to fulfill winning condition *W1* (see Sect. 2.1). We call an SSG transition *implemented* if its event is implemented.

We call a *goal* state implemented if all outgoing transitions labeled with environment events are implemented (cf. condition *W2* in Sect. 2.1). (In goal states, there is no obligation for the system to send messages.)

We call a *non-goal* state *implemented* if it has at least one outgoing transition labeled with a system event that is implemented. Intuitively, there must be at least one message that the system can send. In some cases, it may be that the system can send no message, but there is at least one assumption MSD in an executed cut, i.e., the environment must yet complete a particular sequence of events. In this case, the system can wait for environment events. Hence, a non-goal state is also implemented if there is an assumption MSD in an executed cut and all outgoing transitions labeled with environment events are implemented.

Since we require all reachable states to be implemented, the corresponding strategy does not contain any deadlock state (cf. condition *W1* in Sect. 2.1).

### 3.2 Creation of Successor Candidate

When the algorithm finds any unimplemented SSG states for the current candidate (called $CC$ in the following), it picks any state $s$ of these and creates a successor candidate $CC'$ by copying $CC$. The algorithm attempts to add transitions and states to $CC'$ such that $s$ becomes implemented. These additions depend on whether or not $s$ is a goal state.

**If $s$ is a goal state,** the algorithm checks if the following conditions are met for all unimplemented *uncontrollable* events $e$:

1. No other successor candidate $CC''$ of $CC$ already exists in which $e$ is implemented in $s$.
2. No controller in $CC$ is (already) sending in $s$.

Condition 1 ensures that no candidate for the same combination of $e$ and $s$ is constructed again which was previously found to inevitably lead to a losing strategy. Condition 2 ensures that environment events are considered only in states in which the system does not send anything, because we assume that the system is always faster than the environment.

If both conditions are met, the algorithm adds receiving transitions to the controllers of $CC'$ such that all uncontrollable events in $s$ are implemented. Otherwise, the algorithm handles $s$ in the same way as a non-goal state.

**If $s$ is not a goal state,** the algorithm picks any unimplemented *controllable* event $e$ that fulfills condition 1 and the following conditions instead of condition 2. Condition 3 ensures that controllers remain deterministic, i.e., they send only one event in each state. Condition 4 is necessary to fulfill winning condition *W1* (cf. Sect. 2.1). The algorithm checks condition 4 for $s$ by performing a DFS from the successor state of the SSG transition for $e$ in $s$ via transitions that are implemented for $CC$ and stopping at goal states. Thus the DFS only reaches $s$ if this transition closes a loop without goal state in the strategy.

3. The sender controller $C$ of $e$ does not send another event in $s$.
4. Implementing $e$ in $s$ does not close a loop of implemented SSG transitions without a goal state.

If such an event $e$ exists, the algorithm adds sending transitions to all states of $C$ in $CC'$ that correspond to $s$, such that $e$ is implemented. If no such event exists, the algorithm searches an unimplemented controllable event $e$ that fulfills conditions 3 and 4, *but not 1.* Then, a successor candidate for implementing $e$ was already created, but the DFS backtracked. If $C$ in a state corresponding to $s$ also corresponds to other SSG states $s'$, this backtracking may have been necessary because sending the event $e$ was problematic in $s'$: It can be that, while $e$ must be sent in $s$, sending $e$ in $s'$ violates the specification. Since $C$ cannot distinguish these states, it must send the same message in both of them. The

algorithm then checks whether there is another controller $C'$ that can distinguish at least one such state $s'$ from $s$. If a $C'$ exists, the algorithm adds transitions to the controllers in $CC'$ such that one or several controllers $C'$ send synchronization messages to $C$. These allow $C$ to distinguish $s$ from other states $s'$. If a synchronization was added, $CC'$ is extended as above to make $s$ implemented for $CC'$, without affecting $s'$. Note that we only add synchronization messages when a path via connectors in the CSD (cf. Sect. 2.1) exists such that $C'$ can send messages to $C$. We search such a path via a DFS on the CSD.

If still no such event $e$ could be found, but an assumption MSD is in an executed cut in $s$, the system may wait for the environment in $s$. Then, the algorithm attempts to implement all environment events as discussed above for goal states. In all other cases, the DFS discards $CC$ and backtracks to its predecessor $CC_p$. When attempting to create a new successor for $CC_p$, the same $s$ that was picked when constructing $CC$ as successor of $CC_p$ is selected again. Other SSG states do not need to be considered because if $s$ is not implementable no solution can be reached by extending $CC$.

### 3.3 Removal of Duplicates

After each modification of the CG, the algorithm checks all new candidates and all new controller states for duplicates. It merges the duplicates into one, combining their incoming transitions/edges. We consider candidates as duplicates whenever all their controllers are *identical*. We consider controllers as identical when their sets of states and transitions are identical. Controller transitions are identical if their source and target state and their event are identical. Controller states are identical if they correspond to the same SSG states and if the set of (other) controllers to which synchronization messages have been sent is identical. The latter is necessary because sending a synchronization message must lead to a new controller state—despite identical corresponding states—so that the message will not be sent repeatedly.

### 3.4 Correctness

In the following, we informally argue for the correctness of our algorithm.

**Termination** Our algorithm generally is a DFS on the CG that terminates if the graph is finite. The CG is finite if the number of candidates for the given specification is finite. This, in turn, depends on the maximum number of controller states and transitions. Without considering synchronization transitions, the maximum number of controller states is bounded by the size of the power set of SSG states, as new controller states are created only for new sets of corresponding SSG states. Each synchronization transition adds an additional controller state for the sender controller as it must distinguish the state before sending the message from the state after sending. However, synchronization transitions are only added in cases where a controller needs information from other controllers to send. Thus, in the worst case, for each SSG state an additional state will be

added to all controllers except the one receiving the synchronizations. The number of controller transitions is bounded by the number of controller states and the number of events in the specification, which we assume to be finite. Thus, the total number of candidates depends on the number of system objects and the size of the SSG, which we both also assume to be finite. In conclusion, the CG is finite as well. Computation of the corresponding states requires further DFS runs, but these are performed on the (finite) SSG. Thus, *the algorithm is guaranteed to terminate.*

**Correctness of the resulting solution** If the DFS terminates with a solution, that solution is a candidate with no reachable unimplemented SSG states. The conditions for an SSG state to be implemented directly correspond to the conditions *W1* and *W2* for a winning strategy (cf. Sect. 2.1 and 3.1), except for the requirement to include goal states in circles of implemented transitions in the SSG. However, the algorithm checks for loops without goal states and does not close them by adding transitions to a candidate. Thus, only loops with goal states remain and the strategy defined by all implemented transitions between the reachable states is a winning strategy. Consequently, *the candidate returned by the algorithm is a valid implementation for the given specification.*

## 4 Example Execution of the Distributed Synthesis

We illustrate our algorithm by showing its execution on the example MSD specification presented in Fig. 1 (cf. Sect. 2). Figure 3 shows for several steps of the algorithm snapshots of the candidate controller system $CS$ that has been constructed up to that point, the SSG $S$, and the CG.

**a.1** The snapshot shows the algorithm's models after initialization. The specification defines two system objects: the PressController pC and the ArmController aC. Thus, the algorithm creates initial states for the two controllers, which both correspond to the initial state $S.0$ of the SSG. The initial state is a goal state and the algorithm needs to consider all possible environment events in this state. These are intactBlank and brokenBlank, both leading to new SSG states.

**a.2** The algorithm creates a new candidate which has receiving transitions in aC for brokenBlank and intactBlank, making the SSG states $S.1$ and $S.2$ reachable, but also making $S.0$ implemented. The controller aC can observe both messages and "knows" in which SSG state the system is, either $S.1$ or $S.2$. Thus, the new transitions lead to two new controller states corresponding to one SSG state each. The controller pC, however, cannot observe this message and cannot distinguish $S.1$ and $S.2$ from $S.0$. All three correspond to $pC.0$ (state 0 of $pC$). The algorithm checks for the reachable states $S.0$, $S.1$, and $S.2$, whether they are unimplemented, which is the case for $S.1$ and $S.2$.

**a.3** The algorithm selects $S.1$. Since $S.1$ is not a goal state, the algorithm must pick an enabled system event, which here can only be removeBlank. It adds a transition to aC that sends this event. The algorithm determines that this transition leads back to $S.0$. As all outgoing transitions in $S.0$ are observable for aC, the transition's target state only corresponds to $S.0$. For this set of

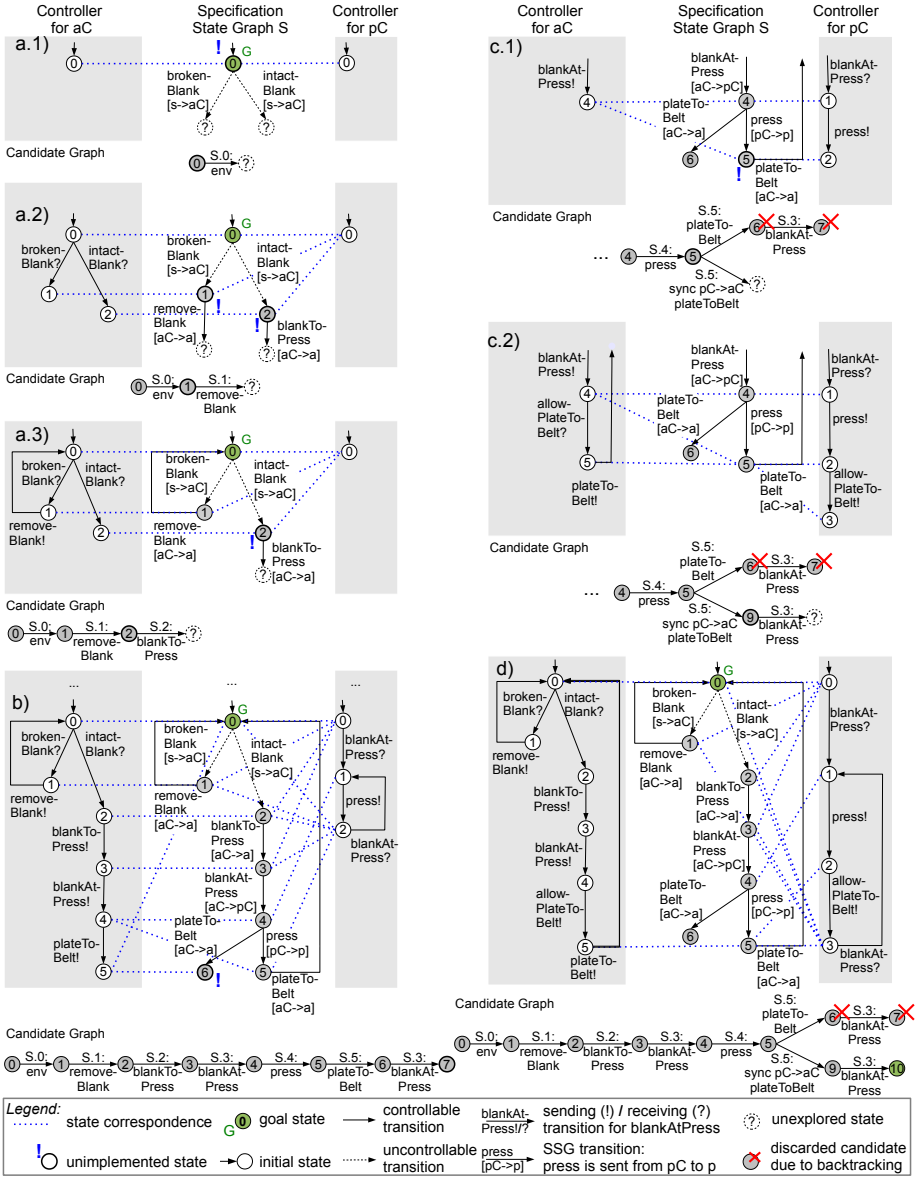


**Fig. 3.** Algorithm steps performed for the example (excerpts)

corresponding states, the controller state $aC.0$ already exists in aC. Thus, the transition leads to $aC.0$. Now, $S.0$ and $S.1$ are implemented, while $S.2$ is not.

**b** The figure then skips several steps until finally, in the step executed before snapshot b, the addition of a controller transition closes the second loop to the goal state $S.0$ in the SSG. Like in step a.3, the algorithm has extended aC to send events as a reaction to an environment event. It sends the second event, blankAtPress, to pC. Hence, the algorithm adds a receiving transition for blankAtPress to pC. The target state $pC.1$ in pC can then distinguish $S.4$ from the other states and sends press. In $S.5$, plateToBelt is sent, which closes the loop to $S.0$, but also makes $S.6$ reachable from $S.4$, because aC.4 corresponds to both, $S.4$ and $S.5$. The closing of the loop to $S.0$ and the subsequent events until blankAtPress are not observable by pC. To compute the corresponding states for $pC.2$, the algorithm performs a DFS starting in $S.0$ via unobservable transitions for pC. Since $S.2$ is among the states found by the DFS, the event blankAtPress was implemented by adding a receiving transition in $pC.2$. However, $S.6$ remains unimplemented.

**c.1** Because $S.6$ turns out to be a deadlock state (due to a safety violation in a requirement MSD) which cannot be implemented, the DFS backtracks until candidate 5 to prevent reachability of $S.6$.

**c.2** The algorithm again attempts to implement $S.5$. The CG shows that immediately sending plateToBelt in $S.5$ was already attempted unsuccessfully. However, it was not yet attempted with prior synchronization to help aC distinguish $S.4$ from $S.5$. The algorithm finds a controller that can distinguish these states, namely pC, since $pC.2$ only corresponds to $S.5$. Therefore, the algorithm adds a synchronization message allowPlateToBelt from pC to aC, telling aC that $S.5$ has been reached and sending plateToBelt is safe now. Consequently, a controller transition sending plateToBelt is added to aC. Note that allowPlateToBelt was not defined in the specification.

**d** After the synchronization, the algorithm will never need to consider plateToBelt in $S.4$ and, thus, never need to consider the deadlock state $S.6$. It just has to implement blankAtPress in $S.3$ as discussed for snapshot b. Then, all reachable SSG states are implemented and the algorithm terminates successfully. Snapshot d shows the final result. We omitted the correspondences of some controller states for reasons of visualization.

## 5 Related Work

Harel et al. [7] describe an approach for controller synthesis from LSCs based on the creation of a product automaton; a distributed implementation is then formed by fully-synchronized copies of a centralized controller. Contrary to our approach, this procedure introduces additional synchronization messages even when the local information of the controllers is sufficient to fulfill the specification. The approach may thus introduce a vast amount of superfluous messages in the generated implementation that degrades its runtime performance.

In later works, Harel et al. [8], and similarly Bontemps et. al. [1], synthesize distributed implementations from LSCs, but they do not add any additional messages not defined in the specification. Therefore, unlike our approach, they cannot handle cases where the local controllers need to share their knowledge about the global state to fulfill the specification.

Sun and Dong [13] present a synthesis approach for LSCs that constructs a distributed implementation in the form of a CSP (Communicating Sequential Processes) system. However, the implementation they create is only valid if the LSC specification already guarantees that, regardless of the behavior of the system, no situation can be reached where liveness requirements cannot be fulfilled without violating safety requirements. Our approach does not have this restriction, because it backtracks in such situations.

While the following approaches do *not* consider LSC/MSD-specifications, they are related as they also model the local knowledge of subsystems.

Halle and Bultan [6] construct local views of subsystems to decide whether a given automata-based protocol can be implemented by a distributed system without additional communication. They mention as possible future work to add additional messages that are required to implement the protocol. However, they do not actually present any such algorithm in the paper.

Finkbeiner and Schewe [5] consider the distributed synthesis problem for specifications using $\omega$-regular tree languages. They take into account the limited knowledge of the synthesized subsystems but, contrary to our approach, they do not consider additional communication to extend this knowledge. Thus, their approach does not work in cases requiring additional communication.

Like our approach, Katz et al. [11] and Peled and Schewe [12] model the local knowledge of subsystems and use synchronizations when this local knowledge is insufficient. However, their approach requires as an input an existing implementation model of a distributed system given as a Petri net. New safety requirements are provided in a textual form. Their algorithm modifies the given system such that these new requirements are fulfilled.

## 6 Conclusion

In this paper, we presented a new approach for synthesizing implementation models for distributed systems based on scenario-based MSD specifications. We presented an algorithm that synthesizes a controller automaton for each subsystem such that they in combination fulfill the specification. The core novelty is that our algorithm keeps track of the local knowledge of the subsystems about the global state. It adds additional synchronizations whenever a subsystem needs to react based on messages that it cannot observe. We demonstrated the application of our approach on an example specification.

We implemented our distributed synthesis algorithm as an Eclipse plug-in based on our ScenarioTools tool suite [3]. We evaluated our approach by executing it on the production cell example presented in the previous sections. Furthermore, we applied it on several variants with a higher number of arms

and presses. We then manually validated that the controllers created by the algorithm are a correct implementation of these MSD specifications.

In future works, we will extend our synthesis algorithm to take into account real-time behavior and asynchronous communication. Our new timed play-out for ScenarioTools [2] paves the way for these extensions. Furthermore, we plan to evaluate the algorithm's performance for larger systems and aim to reduce its runtime by preventing unnecessary re-computation of intermediate results.

## References


1. Bontemps, Y., Heymans, P., Schobbens, P.Y.: Lightweight formal methods for scenario-based software engineering. In: Leue, S., Systä, T. (eds.) Scenarios: Models, Transformations and Tools, Int. Workshop, Dagstuhl Castle, Germany, Revised Selected Papers. LNCS, vol. 3466, pp. 174–192. Springer (2003)
2. Brenner, C., Greenyer, J., Holtmann, J., Liebel, G., Stieglbauer, G., Tichy, M.: Scenariotools real-time play-out for test sequence validation in an automotive case study. In: Proc. of 13th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'14) (2014)
3. Brenner, C., Greenyer, J., Panzica La Manna, V.: The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In: Proc. of 12th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'13) (2013)
4. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. In: Formal Methods in System Design. vol. 19, pp. 45–80. Kluwer (2001)
5. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proc. of 20th IEEE Symp. on Logic in Computer Science. pp. 321–330 (2005)
6. Halle, S., Bultan, T.: Realizability Analysis for Message-based Interactions Using Shared-State Projections. In: Proc. of 18th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering, FSE 2010, Santa Fe, New Mexico (2010)
7. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. Foundations of Computer Science 13:1, 5–51 (2002)
8. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: Generating statechart models from scenario-based requirements. In: Formal Methods in Software and Systems Modeling. vol. 3393, pp. 309–324. Springer (2005)
9. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software and Systems Modeling (SoSyM) 7(2), 237–252 (2008)
10. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (2003)
11. Katz, G., Peled, D., Schewe, S.: Synthesis of distributed control through knowledge accumulation. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification, LNCS, vol. 6806, pp. 510–525. Springer (2011)
12. Peled, D., Schewe, S.: Practical distributed control synthesis. In: Yu, F., Wang, C. (eds.) Proc. Int. Workshop on Verification and Infinite State Systems (INFINITY'11). EPTCS, vol. 73, pp. 2–17 (2011)
13. Sun, J., Dong, J.S.: Synthesis of distributed processes from scenario-based specifications. In: Proc. of 2005 Int. Conf. on Formal Methods. pp. 415–431. FM'05, Springer-Verlag, Berlin, Heidelberg (2005)