Compositional Synthesis of Controllers from Scenario-Based Assume-Guarantee Specifications

Joel Greenyer[1], Ekkart Kindler[2]

[1] Software Engineering Group, Leibniz Universität Hannover, Germany
greenyer@inf.uni-hannover.de

[2] DTU Compute, Technical University of Denmark, Denmark
ekki@dtu.dk

Abstract. Modern software-intensive systems often consist of multiple components that interact to fulfill complex functions in sometimes safety-critical situations. During the design, it is crucial to specify the system's requirements formally and to detect inconsistencies as early as possible in order to avoid flaws in the product or costly iterations during its development. We propose to use Modal Sequence Diagrams (MSDs), a formal, yet intuitive formalism for specifying the interaction of a system with its environment, and developed a formal synthesis approach that allows us to detect inconsistencies and even to automatically synthesize controllers from MSD specifications. The technique is suited for specifications of technical systems with real-time constraints and environment assumptions. However, synthesis is computationally expensive. In order to employ synthesis also for larger specifications, we present, in this paper, a novel assume-guarantee-style compositional synthesis technique for MSD specifications. We provide evaluation results underlining the benefit of our approach and formally justify its correctness.

Keywords: Scenario-Based Specification, Compositional Controller Synthesis, Consistency Checking, Assume-Guarantee

1 Introduction

Modern software-intensive systems in areas like transportation or production often consist of many components that interact to provide complex functionality in sometimes safety-critical situations. In the early design, interactions are typically specified by scenarios. We propose a model-based approach and use Modal Sequence Diagrams (MSDs), introduced by Harel and Maoz [8], to specify interaction scenarios. MSDs are a formal interpretation of UML sequence diagrams, based on the concepts of Live Sequence Charts (LSCs) [6], and allow engineers to specify which sequences of events may, must, or must not happen in a system that reacts to events in its environment. We extended MSDs to support real-time constraints and assumptions on the environment. These extensions are important for the specification of mechatronic systems where the software interacts with physical/mechanical parts of the system. Furthermore, we developed

a technique for synthesizing controllers from such specifications and for showing their consistency [7].

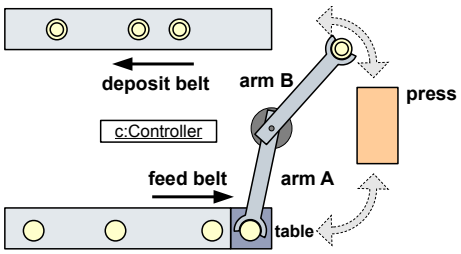
Formal scenario-based modeling and synthesis techniques have the potential to immensely aid engineers in the development of modern technical systems, but unfortunately, synthesis is computationally complex. To make synthesis feasible also for bigger specifications, we present, in this paper, a novel technique that for certain kinds of specifications, allows engineers to decompose the synthesis problem into two parts that can be solved more efficiently.

The technique comprises of four manual steps that require the engineer to (1) subdivide the component structure of the system in two parts, (2) possibly splitting components in two, and (3) subdividing the MSD specification accordingly. Last (4), additional MSDs may be introduced as additional requirements that one part of the system can assume about the other, in order to help it realize its part specification. If controllers can be successfully synthesized for the resulting part specifications, the composition of these controllers forms an implementation of the overall specification. We present a formal justification for the soundness of our technique, which was inspired by Stark [18].

The technique presented in this paper is the first that allows for the decomposition of the synthesis problem for LSC/MSD specifications into two synthesis tasks that can be solved independently. Kugler and Segall also proposed a compositional synthesis approach for LSC specifications [13] that improves the synthesis' efficiency. In their approach, controllers are also synthesized for specification parts. Ultimately, however, always a last synthesis step is required to obtain a controller for the complete specification from the controllers for the specification parts. This is not the case with our technique, and thus we can often more drastically reduce the complexity of the synthesis problem. Also Maoz and Sa'ar recently proposed a technique for synthesizing controllers from LSC specifications with environment assumptions [16], but they do not address the decomposition of the synthesis problem.

Our technique requires the creativity of the engineer in finding a viable decomposition of the specification as well as assume/guarantee properties that are small enough so that the compositional synthesis is of advantage. If, for a chosen decomposition of the specification, no controllers could be synthesized, this does not imply that there does not exist a controller for the global specification. There might be other decompositions for which synthesizing controllers was possible. In this sense, our technique is not complete. Another limitation of our approach is that, in the decomposition, the second part specification can make assumptions about the first, but not vice versa. Supporting assumptions in both directions would require extra mechanisms, which we plan to investigate in future work.

This paper is structured as follows. We introduce an example in Sect. 2 and explain the foundations in Sect. 3. We describe our compositional synthesis technique in Sect. 4. Here we focus mainly on the technical aspects of creating a correct specification decomposition, but give a brief discussion on the methodology for using our technique. We then present realization details and evaluation results in Sect. 5, discuss related work in Sect. 6, and conclude in Sect. 7.



Environment Assumptions:

A1) The interarrival time of plates on the table is greater FMIN.

A2) The time for moving arm A from the table to the press or from the press to the table is between AMIN and AMAX.

A3) The time for moving arm B from the press to the deposit belt or from the deposit belt to the press is between BMIN and BMAX.

A4) The time for the press to press a blank plate is between PMIN and PMAX.

Requirements:

R1) When a blank plate arrives at the table, arm A must pick it up, move it to the press, and release it into the press. The arm then has to move back to the table, where it must arrive before the next blank arrives.

R2) When arm A has released the blank into the press, the press must press it, and then arm B must pick up the plate.

R3) When arm B has picked up the processed plate, it must transport the plate to the deposit belt.

R4) When arm B has arrived at the deposit belt, it must release the processed plate and then move back to the press.

R5) Arm A must only release a blank into the press if arm B has picked up the processed plate from the press.

R6) Arm A must not attempt to pick up the next blank before having returned to the table.

R7) Arm B must not attempt to pick up the pressed plate before having returned to the press.

Fig. 1. A sketch of the production cell system and its textual specification

2 Example

As an example, we consider a simplified specification of a production cell [14], an industrial production robot with two arms. One arm, *arm A*, picks up metal blanks that arrive from a *feed belt* on a *table* and places them into a *press*, where they are pressed into plates. The other arm, *arm B*, picks up the pressed plates and places them on a *deposit belt*, where they are transported off again. Figure 1 shows the system with its requirements and environment assumptions in plain text. Initially, arm A is located at the table, and arm B is located at the press.

After formalizing the above requirements and assumptions into an MSD specification MS for a single controller component $\underline{\mathsf{c}}$, the technique described in this paper will allow us to

1. split the controller component $\underline{\mathsf{c}}$ into two components, $\underline{\mathsf{c1}}$ for arm A, and $\underline{\mathsf{c2}}$ for arm B and the press,
2. split MS into two part specifications MS_1 and MS_2 for $\underline{\mathsf{c1}}$ resp. $\underline{\mathsf{c2}}$,
3. introduce additional properties as assumptions to MS_2 and as requirements to MS_1, with the aim of helping $\underline{\mathsf{c2}}$ in being able to realize MS_2, while not making it impossible for $\underline{\mathsf{c1}}$ to realize MS_1,

so that finally, if controllers $\underline{\mathsf{c1}}$ and $\underline{\mathsf{c2}}$ can be synthesized, they together form an implementation for the global specification MS.

3 Foundations

As foundations, we first formalize a notion of components that interact via messages. Then we introduce controllers and MSDs. As time is relevant in our example, we consider a timed setting. Our technique, however, is also applicable in an untimed setting.

3.1 Object Systems, Message Events, Runs

We consider systems of *objects* that interact via messages. Our definitions are based on Harel and Marelly [9]. For brevity, we consider synchronous messages only. Our technique would in principle also work for asynchronous communication, but this would need to be formalized.

Definition 1 (Object system, message, messages event, alphabet). *An* object system *consists of a set of objects O that exchange* messages*. A message has a name (from a set of names $Name$) and a sending and receiving object. The sending and receiving of a message in an object system is a single* message event*, or simply* event*, $e \in O \times Name \times O$. The set of possible message events is called the* alphabet*, denoted with $\Sigma \subseteq O \times Name \times O$.*

We consider a timed setting where message events occur at certain points in time. The progress of time is represented by a sequence of positive, increasing real values [1]. A message event itself does not take any time.

Definition 2 (Timed event, timed words, timed language). *A* timed event *is a pair $(e, r) \in (\Sigma \times \mathbb{R}^{\geq 0})$ where e is a message event occurring at time r. A* timed word *$\pi \in (\Sigma \times \mathbb{R}^{\geq 0})^{\omega}$ is an infinite sequence of timed events. For every two subsequent timed events in a timed word $\pi = \ldots, (e_i, r_i), (e_{i+1}, r_{i+1}), \ldots$, we require that $r_i \leq r_{i+1}$. Furthermore, for every $r \in \mathbb{R}^{\geq 0}$, we require that there exists a timed event (e_i, r_i) such that $r_i > r$, i.e., time must progress. The set of all timed words is denoted by $\mathcal{L}$ and a subset $L \subseteq \mathcal{L}$ is called a* timed language*. The complement of a language L is denoted by $\overline{L}$ and defined as $\overline{L} = \mathcal{L} \setminus L$.*

3.2 Controllers and Parallel Composition

Subsets of objects in the object system can be controlled by a *controller*. There can be multiple controllers, but the objects controlled by different controllers must be disjoint. We consider a controller to be a timed automaton [1] with some additional constraints, which will be described shortly. We rely on the usual definitions [1,5], and only briefly and informally explain the essential concepts, since the concrete controller formalism is not important for our approach.

A *timed automaton* $TA = (\Sigma, S, S_0, X, I, T)$ is an automaton with a finite set of *locations* S, $S_0 \subseteq S$ being *start locations*, and a finite set of real-valued variables X, called *clocks*, which increase synchronously and monotonically over time. I are *invariants* for locations, which specify that a timed automaton must not be in a location at a certain time. A timed automaton has *edges* between locations, which are defined through a relation $T \subseteq S \times \Sigma \times \mathcal{C}(X) \times 2^X \times S$. An edge $(s_s, e, \psi, \lambda, s_t)$ goes from location s_s to location s_t and is labeled by an event e. The element $\psi \in \mathcal{C}(X)$ is called a *constraint* on clock variables that is the *guard* of an edge, permitting it to be taken only at certain times. $\lambda \in 2^X$ is a subset of clocks that are *reset* when the edge is taken. A timed automaton *accepts* a timed language. For an automaton TA, we denote the accepted timed language as $L(TA)$.

If two timed automata share some events but have a disjoint set of clocks, we can form the *parallel composition* of the two automata, which is defined through the construction of the product of the two automata. For two timed automata TA_1 and TA_2, the parallel composition is denoted as $TA_1||TA_2$.

For a controller of a subset of objects in the object system, we require that for each message event not sent or received by an object that is controlled by the controller, each location has unguarded self-edges labelled with that event and without any clock reset. This requirement reflects the fact that a controller for a particular subset of objects should not be able to block the sending or receiving of messages among objects that it does not control.

Definition 3 (Controller). *We define a controller C as an extended timed automaton: $C = (\Sigma, S, S_0, X, I, T, O_C)$, where $O_C \subseteq O$ is a subset of objects in an object system O* controlled *by C. Let $\Sigma_C = \Sigma \cap ((O_C \times Name \times O) \cup (O \times Name \times O_C))$ be the messages sent and received by the objects in O_C. Then for every controller C we require that for every location $s \in S$ and every event $e \in \Sigma \setminus \Sigma_C$, there is an edge $(s, e, true, \emptyset, s) \in T$ (unguarded, no clocks reset). If $C1$ and $C2$ are controllers for objects O_{C1} and O_{C2}, we require that $O_{C1} \cap O_{C2} = \emptyset$.*

The additional self-edge for all the message events sent and received by the object not controlled by a controller allow us to infer the following.

Lemma 1 (Composition is conjunction). *Let C_1 and C_2 be two controllers accepting the languages $L(C_1)$ and $L(C_2)$. Then $L(C_1||C_2) = L(C_1) \cap L(C_2)$.*

We assume an open-world setting where the object system is subdivided into *system objects* and *environment objects*. System objects are *controllable*, i.e., the objects we seek an implementation for. Environment objects are *uncontrollable*; they represent for example sensors and actuators by which a software controller monitors and acts upon the physical world.

In this setting, we assume that if an environment event occurs, the system objects can immediately take any finite number of steps to react to this event before the next environment event occurs (in accordance with Harel and Marelly [9]). If the system waits for time to pass, environment events may occur again. This behavior can be ensured by formulating certain restrictions for the controllers of the environment and system objects, but we omit these restrictions for brevity.

3.3 MSD Specifications

An MSD specification specifies the valid interaction behavior in an object system [8]. We consider MSD specifications that not only formulate requirements on the system, but also formulate assumptions on how the environment behaves. The requirements and assumptions are two sets of MSDs.

Furthermore, MSDs can be *existential* or *universal*. Existential diagrams specify sequences of events that must be possible to occur in the object system and universal diagrams specify requirements that must be satisfied by all the sequences of events. We consider only universal MSDs in this paper.

MSDs, lifelines, and messages An MSD is a sequence diagram where each lifeline represents an object in the object system. A message in an MSD represents a message event. Furthermore, a message has a *temperature* and an *execution kind*. The temperature can be either *hot* or *cold*; the execution kind can be either *monitored* or *executed*. Figure 2 shows two MSDs ArmATransportBlankToPress and PressPlateAfterArmAReleasesBlankPlate from the production cell specification. They formalize the requirements $R1$ and $R2$ and refer to the object system sketched at the top of Fig. 1. The temperature and execution kind are indicated by a label (e.g., h,c, e,m). The hot or cold temperature is also represented by red or blue color of the arrows. Monitored messages also have a dashed arrow; executed messages have a solid arrow.

Intuitively, a monitored message says that something may happen whereas an executed message says that something must eventually happen (liveness). A hot message says that no event expected at another point in the scenario must occur (safety) before the event represented by that message occurs. A cold message, by contrast, says that this may happen. We assume that an MSD has only one first message, which must be cold and monitored.

For example, consider the MSD ArmATransportBlankToPress in Fig. 2: The hot and executed message `pickUp` says that after `blankArrived` occurred, `pickUp` must eventually occur and no other event represented by a message in the diagram is allowed to occur. Then, likewise, `moveToPress` must occur. The hot, but monitored message `arrivedAtPress` means that `arrivedAtPress` may occur, but as long as it does not occur, no other message event in the diagram must occur. Then `releaseBlank` must occur, etc. This interpretation of the message temperature and execution kind extends the original definition [8] where the temperature alone reflects both the safety and liveness requirements.

More specifically, the semantics of these messages is as follows: When an event occurs in the system that is represented by the first message in an MSD, an *active copy* of the MSD or *active MSD* is created. As further events occur that are represented by the subsequent messages in the diagram, the active MSD progresses. This progress is captured by the *cut*, which marks for every lifeline the locations where the occurred messages are attached to the lifeline. If the cut reaches the end of an active MSD, the active copy is terminated.

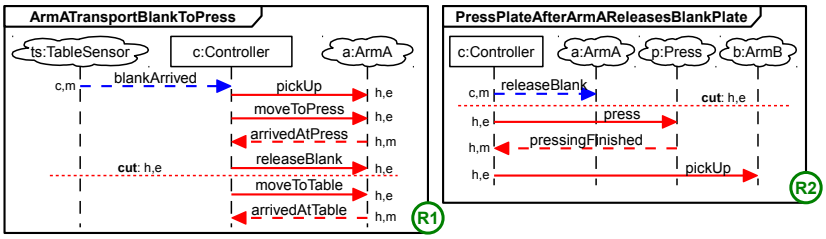


Fig. 2. The MSDs of the production cell specification for requirements R1 and R2

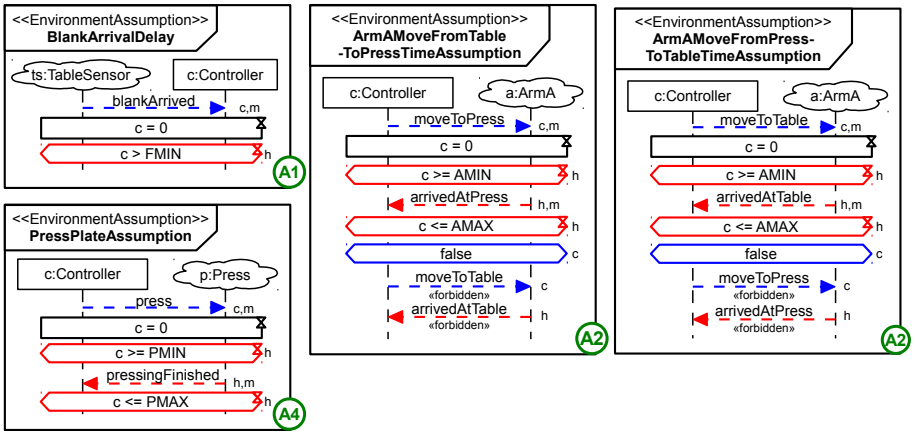


Fig. 3. The MSDs of the production cell specification for assumptions A1, A2, and A4

If the cut is in front of a message on its sending and receiving lifeline, the message is *enabled*. If a hot message is enabled, the cut is also *hot*; otherwise the cut is *cold*. Similarly, if an executed message is enabled, the cut is also *executed*; otherwise the cut is *monitored*.

A *safety violation* occurs if, in a hot cut, a message event occurs that is represented by a message in the MSD that is not currently enabled. If the same situation occurs in a cold cut, it is called a *cold violation*. Safety violations must never happen, while cold violations may occur and result in terminating the active copy of the MSD. If the cut is executed, this means that the active MSD must progress and it is a *liveness violation* if an active MSD never terminates or progresses to a monitored cut.

There can be multiple active copies of MSDs at a time. Figure 2 shows a reachable configuration of cuts for (active copies of) the MSDs ArmATransportBlankToPress and PressPlateAfterArmAReleasesBlankPlate.

Environment assumptions, time, and forbidden messages We model environment assumptions by MSDs that have an additional label «EnvironmentAssumption». Figure 3 shows assumption MSDs from the production cell example. BlankArrivalDelay models the assumption $A1$. The MSDs ArmAMoveFromPressToTableTimeAssumption and ArmAMoveFromTableToPressTimeAssumption model the assumptions $A2$, and the MSD PressPlateAssumption models the assumption $A4$. The MSDs modeling the assumption $A3$ are very similar to those modeling assumption $A2$, and are thus omitted. In these MSDs, we find additional constructs, namely *clock resets*, *conditions*, and *forbidden messages*.

Time constraints can be modeled in MSDs with resets of real-valued clock variables and conditions, similar to timed automata. Clock resets and conditions are boxes resp. hexagons that span one or multiple lifelines. If the cut is immediately before a clock reset or condition on all the lifelines it spans, the clock reset

or condition is *enabled*. If a clock reset is enabled, then immediately, and before any other message event occurs, the clock variable is reset to zero and the cut progresses beyond the clock reset.

Conditions have a temperature (hot or cold), represented by a red resp. blue border color. In our figures, they have an additional label (h/c). We distinguish *timed* and *untimed* conditions. In this paper, untimed conditions have only the expression *true* or *false*. Timed conditions have an attached hour-glass symbol and can have expressions of the form $x \bowtie expr$ where x is a clock variable, *expr* is an integer constant, and $\bowtie$ is an operator $<, \leq, >, \geq$.

If a condition is enabled, and its expression evaluates to true, the cut progresses immediately and before any other message event occurs. If the expression of a cold condition evaluates to false, the active MSD is terminated. If the expression of a hot condition evaluates to false, the cut cannot progress, but at the same time it is a liveness violation if the cut never progresses. From this follows that it is a liveness violation if a hot untimed *false* condition is enabled.

For hot timed conditions, we distinguish *minimal delays* ($\bowtie \in \{>, \geq\}$) and *maximal delays* ($\bowtie \in \{<, \leq\}$). If a minimal delay evaluates to false, the cut progresses as soon as it becomes true. Meanwhile the cut is hot, i.e., no message that is not currently enabled in the active MSD is allowed to occur. If a maximal delay evaluates to false, this is a liveness violation of the MSD.

In the MSD BlankArrivalDelay, for example, a clock reset followed by a minimal delay is used to formalize the assumption that blanks arrive on the table with a certain minimal delay: after `blankArrived` occurred, the clock c is immediately reset to zero and then the minimal delay will be enabled until $FMIN$ time units have passed. In this time `blankArrived` must not occur.

At the end of an MSD, separated by a terminal cold *false* condition, there can be hot or cold *forbidden messages*. If there is an active MSD and a message event occurs that is represented in the MSD by a cold forbidden message, this is a cold violation, and the active MSD terminates. If a message event occurs that is represented by a hot forbidden message, this is a safety violation.

In the MSD ArmAMoveFromTableToPressTimeAssumption a clock reset and hot time conditions are used to express that after `moveToPress`, the event `arrivedAtPress` must occur within a certain interval. The hot forbidden message states that, in this interval, also `arrivedAtTable` must not occur. The cold forbidden message `moveToTable` states that `moveToTable` is allowed to occur, but, since this leads to the termination of the active MSD, then it cannot be assumed that the arm will arrive at the press in the specified interval.

To complete the example, Fig. 4 shows the MSDs for the requirements $R3$-$R7$.

Satisfying and implementing a specification, and consistency Without giving a more formal definition on the MSD semantics, we denote the (timed) language accepted by an MSD D as $L(D)$.

Definition 4 (Language accepted by a set of MSDs). *For an object system O and a set of message events Σ, Let $M = \{D_1, \ldots, D_n\}$ be a finite set of MSDs. We define the language* accepted *by M as $L(M) = \bigcap_{i=1}^{n} L(D_i)$.*

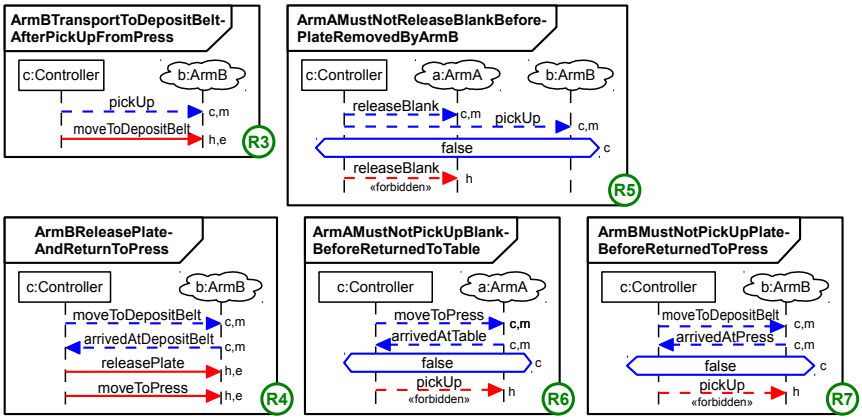


Fig. 4. The MSDs of the production cell specification for requirements R3–R7

A (timed) word *satisfies* an MSD specification if it is accepted by the requirement MSDs or not accepted by the assumption MSDs[3].

Definition 5 (Satisfying an MSD specification). *For an object system O and a set of message events Σ, $MS = (A, G, O_E, O_S)$ is an MSD specification where A and G are sets of MSDs. A is called the* assumptions *and G is called the* requirements *or* guarantees. *O_E are the environment objects and O_S are the system objects, $O_E \cup O_S = O$, $O_E \cap O_S = \emptyset$. The language* satisfying *MS, denoted with $L(MS)$, is defined as $L(MS) = \overline{L(A)} \cup L(G)$. A controller C for all objects O* satisfies *an MSD specification MS iff $L(C) \subseteq L(MS)$.*

A system controller for all the system objects *implements* an MSD specification if the controller that results from the composition with any possible environment controller satisfies the specification.

Definition 6 (Implementing an MSD specification, consistency). *Given an MSD specification $MS = (A, G, O_E, O_S)$, a system controller C_S for O_S* implements *MS if, for the closed system formed by the composition with every possible environment controller C_E for O_E holds $L(C_E||C_S) \subseteq L(MS)$. From Lemma 1 and our definition of environment controllers, this is equivalent to $L(C_S) \subseteq L(MS)$. An MSD specification is* consistent *if there exists a system controller for all the system objects O_S that implements the specification.*

4 The Assume-Guarantee Synthesis Approach

Given an MSD specification MS, called the *global specification* in the following, this section explains how to decompose this specification into two specifications

[3] Of course, we expect the environment to satisfy the assumptions, but in environments that do not, the system is not required to satisfy the requirements.

MS_1 and MS_2, called *part specifications*, possibly adding MSDs as requirements to MS_1 and assumptions to MS_2, so that the consistency of the global specification follows from the consistency of the part specifications.

4.1 Decomposing the Global Specification

Assume a given MSD specification $MS = (A, G, O_E, O_S)$ for the objects O. Then, we can decompose this specification as follows.

Step 1 (Subdivide the set of system objects). We subdivide the objects O_S into two disjoint sets O_{S1} and O_{S2} with $O_{S1} \cup O_{S2} = O_S$ and $O_{S1} \cap O_{S2} = \emptyset$ with the respective environment objects $O_{E1} = O \setminus O_{S1}$ and $O_{E2} = O \setminus O_{S2}$.

Step 2 (Create subsets of MSDs for the part specifications). We create two part specifications $MS_1 = (A_1, G_1, O_{E1}, O_{S1})$ and $MS_2 = (A_2, G_2, O_{E2}, O_{S2})$ such that $G_1 \cup G_2 = G$ and each part specification may contain any subset of the assumption MSDs in the global specification: i.e. $A_1, A_2 \subseteq A$.

If we can now successfully synthesize system controllers from the part specifications, this means that these controllers implement their part specification regardless of the behavior of their opposite controller. It may be, however, that one controller must assume additional properties about the other controller, which, in turn, must guarantee these properties.

Step 3 (Add assume/guarantee properties to the part specifications). A set of MSDs, called AG^+, can be added as additional assumptions to part specification MS_2 and as additional requirements to part specification MS_1, i.e., $MS_1 = (A_1, G_1 \cup AG^+, O_{E1}, O_{S1})$ and $MS_2 = (A_2 \cup AG^+, G_2, O_{E2}, O_{S2})$

Now the implementation of the second part specification makes assumptions on the implementation of the first. Currently, we allow this only in one direction. Otherwise, we could always easily construct two part specifications that are consistent only because both controllers can mutually violate their assumptions, but then fail to implement the global specification.

4.2 Decomposing System Objects

In many cases, as in our production cell, it is necessary to decompose a system object into two objects that fulfill distinct functions in the two part specifications. This requires also to change the MSD specification such that for the two resulting objects an equivalent behavior is specified as for the initial object.

The goal is to split up the modified specification in such a way that one part of the specification specifies the behavior of the first object and another part of the specification specifies the behavior of the second object. In order to successfully apply the described compositional synthesis technique, the behavior of the first object must be independent from the behavior of the second, i.e., there may remain MSDs that specify how the second object must react to events involving the first object, but not vice versa. An example follows in Sect. 4.3.

Decomposing a system object is done before Step 1; thus, we call it “Step 0”:

Step 0 (Decomposing system objects). An object that is a system object in the global specification can be decomposed into two system objects. This implies the following changes:

1. The events that the initial object sends and receives have to be separated into the message events that the resulting objects send and receive.
2. In each MSD of the specification where a lifeline represents the initial object, this lifeline must be split into two lifelines that represent the two objects resulting from the decomposition.
3. Also the diagram messages attached to the original lifelines must be attached to one of the resulting lifelines according to the changed message events. In MSDs where the effect is that one of the lifelines does not send or receive any messages, this lifeline can then be removed. Otherwise,
4. the lifelines must be synchronized so that the order of the events as in the original MSD is preserved. This can be achieved by introducing conditions with the expression `true` that cover both lifelines. These conditions must always be introduced between two messages where a message attached to one lifeline is followed by a message attached to the other lifeline. (We assume that the specified synchronization can always be realized in the final implementation.)

4.3 The Decomposition of the Production Cell Specification

For our production cell example, we first decompose the system object c into the objects c1 and c2 as already explained in Sect. 2: c1 interacts with the table sensor and arm A, c2 with the press and arm B. With an according separation of message events, the MSDs shown previously must be altered as follows:

1. The MSDs representing the requirements $R1$ and $R6$ and assumptions $A1$ and $A2$ are changed so that the lifeline representing object c is replaced by one lifeline representing the object c1.
2. The MSDs representing the requirements $R3$, $R4$, and $R7$ as well as the assumptions $A3$ and $A4$ are changed so that the lifeline representing object c is replaced by one lifeline representing the object c2.
3. The MSDs for the requirements $R2$ and $R5$ are replaced as shown in Fig. 5.

The part specifications are created such that c1 is the system object in the first part specification and c2 is the system object in the second part specification. Then the MSDs (as modified in Step 0) are split up so that the first part specification is made up of the requirements MSDs for $R1$ and $R6$ and the assumption MSDs for $A1$ and $A2$. The second part specification is made up of the MSDs for $R2$ – $R5$, $R7$, and $A3$ and $A4$.

Finally, the MSD BlankArrivalAtPressDelay as shown in Fig. 6 is added as an additional assumption to the second part specification and as an additional requirement to the first part specification. The MSD specifies that controller c1 must order arm A to release blank plates into the press with a minimal time delay of $RMIN$. For certain values for $RMIN$ and the other constants, Sect. 5 documents the results of the controllers synthesized for the global specification and the part specifications.

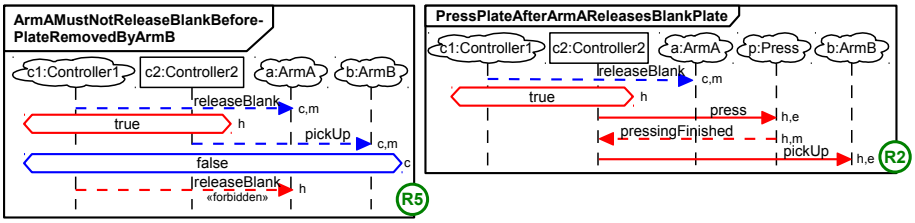


Fig. 5. The MSDs for the requirements R2 and R5 after decomposing the controller

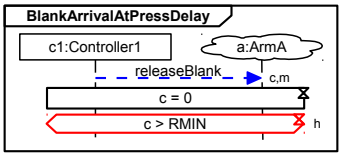


Fig. 6. The MSDs representing the additional assume-guarantee property

4.4 Soundness of the Compositional Synthesis Technique

For proving the soundness of the composition, we assume that we have specifications MS, MS_1 and MS_2 as defined in Sect. 4.1. And we assume that we have two system controllers C_1 and C_2 which implement MS_1 and MS_2 respectively:

$$L(C_1) \subseteq L(MS_1) \tag{1}$$
$$L(C_2) \subseteq L(MS_2) \tag{2}$$

In order to show $L(C_1||C_2) \subseteq L(MS)$, we use the following properties, which can be derived from Def. 4, 5, and 6, and the definition of MS, MS_1 and MS_2:

$$L(MS_1) = \overline{L(A_1)} \cup (L(G_1) \cap L(AG^+)) \tag{3}$$
$$L(MS_2) = \overline{L(A_2)} \cup \overline{L(AG^+)} \cup L(G_2) \tag{4}$$
$$L(MS) = \overline{L(A_1)} \cup \overline{L(A_2)} \cup (L(G_1) \cap L(G_2)) \tag{5}$$

Combining these properties we obtain

$$\begin{aligned}
&L(C_1||C_2) && = \text{by Lemma 1}\\
&L(C_1) \cap L(C_2) && \subseteq \text{by (1) and (2)}\\
&L(MS_1) \cap L(MS_2) && = \text{by (3) and (4)}\\
&(\overline{L(A_1)} \cup (L(G_1) \cap L(AG^+))) \cap && \\
&\quad (\overline{L(A_2)} \cup \overline{L(AG^+)} \cup L(G_2)) && = \text{laws of boolean algebra}\\
&(\overline{L(A_1)} \cap \overline{L(A_2)}) \cup (\overline{L(A_1)} \cap \overline{L(AG^+)}) \cup && \\
&\quad (\overline{L(A_1)} \cap L(G_2)) \cup && \\
&\quad (L(G_1) \cap L(AG^+) \cap \overline{L(A_2)}) \cup && \\
&\quad (L(G_1) \cap L(AG^+) \cap \overline{L(AG^+)}) \cup && \\
&\quad (L(G_1) \cap L(AG^+) \cap L(G_2)) && \subseteq \text{laws of boolean algebra}\\
&\overline{L(A_1)} \cup \overline{L(A_2)} \cup (L(G_1) \cap L(G_2)) && = \text{by (5)}\\
&L(MS) &&
\end{aligned}$$

4.5 Methodology

Once an engineer has decomposed a specification in such a way that the technical conditions of Sect. 4.1 are met, the presented approach is fully automatic. The question remains how an engineer can come up with such a decomposition. This is a question of methodology, which we can only address briefly here.

We argue that engineers who design a complex system typically have a good idea of how to split up the system in order to keep on top of its complexity. The split into components would follow this "mindset".

Our approach works for systems where we can identify components that build on each other, so that later components can make assumptions on earlier ones, avoiding cyclic assumptions. The additional assume-guarantee properties $AG+$ can be used to restrict the timing or relative order of message events that one component shares with an other.

5 Realization and Evaluation

In ScenarioTools[4] [7], we implemented a synthesis technique for timed MSD specifications by a mapping from a UML-based MSD specifications in Eclipse to Timed Game Automata (TGA) that are input for Uppaal Tiga [3,2], an extension of the Uppaal model checker for solving two-player games.

With this mapping, we formulate a winning condition that checks whether there is a strategy for the system to always eventually reach a state where all cuts of requirement MSDs are monitored and no safety violation occurred in any requirement MSDs, or there is an executed cut in an assumption MSD, or a safety violation occurred in an assumption MSD. We call this the $AGAF$ condition. We also check a weaker condition for which strategies can be synthesized more quickly: here we only check that never safety violations occur in requirement MSDs or they occur in assumption MSDs. We call this the AG condition. If the winning condition is satisfied, the tool generates a winning strategy for the system; if not, a winning strategy for the environment is generated. From the winning strategy, a controller can be derived.

In our MSD-to-TGA mapping, we can also specify different degrees of freedom for the system: Either it can always choose to send any system message and also consider to wait for environment events, or we can restrict it to immediately send only system messages that correspond to an executed message in a requirement MSD that is enabled in a current cut. The latter corresponds to the behavior of the *play-out* algorithm, an executable semantics for LSCs/MSDs [10,15], and can drastically simplify the synthesis. If a strategy could be synthesized in the latter setting, the specification is called *consistently executable*.

Figure 7 shows the synthesis times and results from checking the consistent executability of the part specifications and the global specification for the production cell example with different values for $FMIN$, $RMIN$, etc. $RMIN$ is irrelevant for the global specification, since it was only added with the MSD

[4] `http://www.cs.uni-paderborn.de/index.php?id=scenariotools`

Times measured in seconds on a machine with 5 Intel Xenon E5520 CPUs at 2,27Ghz and 72 GB RAM, Suse Linux (only one core and 4GB RAM used by Uppaal Tiga)

	FMIN	RMIN	AMIN	AMAX	BMIN	BMAX	PMIN	PMAX	Part spec. I consistently executable?	Time for synth. AG cond.	Time for synth. AGAF cond.	Part spec. II consistently executable?	Time for synth. AG cond.	Time for synth. AGAF cond.	Global spec. consistently executable? (ignoring values for RMIN)	Time for synth. AG cond.	Time for synth. AGAF cond.
1	8	7	2	3	2	3	1	2	yes	0,13	0,68	yes	0,3	0,44	yes	2,36	5,17
2	**0**	7	2	3	2	3	1	2	no	0,16	1,06	yes	(as in 1)	(as in 1)	no	0,63	71,42
3	**1**	7	2	3	2	3	1	2	no	0,16	0,88	yes	(as in 1)	(as in 1)	no	0,91	82,08
4	**5**	7	2	3	2	3	1	2	no	0,15	1,1	yes	(as in 1)	(as in 1)	no	4,32	84,3
5	8	7	2	**4**	2	3	1	2	no	0,14	1,42	yes	(as in 1)	(as in 1)	no	2,94	69,69
6	8	7	2	3	2	**4**	1	2	yes	(as in 1)	(as in 1)	no	0,32	2,73	no	3,14	77,35
7	8	7	2	3	2	4	1	**6**	yes	(as in 1)	(as in 1)	no	0,35	2,84	no	4,84	105,35
8	8	7	2	3	2	4	1	**8**	yes	(as in 1)	(as in 1)	no	0,35	2,49	no	5,87	123,69
9	8	**3**	2	3	2	3	1	2	yes	0,13	0,59	no	0,4	2,68	--	--	--
10	8	**9**	2	3	2	3	1	2	no	0,14	1,3	yes	0,3	0,43	--	--	--

Fig. 7. The synthesis times for the part specifications vs. the global specification of the production cell with different values for the constants $FMIN$, $RMIN$, etc.

PressPlateAssumption, see Fig. 6. The table shows that in the case of consistent constant values the sum of the time needed to synthesize a strategy for the part specifications (0.68 seconds + 0.44 seconds = 1.12 seconds) is only one fifth of the time needed for synthesizing a strategy for the global specification (5.17 seconds). For more evaluation results and discussion, see [7, Appendix C].

6 Related Work

Our technique is the first that allows for the decomposition of the synthesis problem for LSCs/MSDs into two problems that can be solved independently.

Kugler and Segall also proposed a compositional approach for synthesizing controllers from LSC specifications [13]. With their approach, however, the synthesis problem cannot be split into two separate parts. They first do synthesize controllers for subsets of LSC in a specification—the resulting controllers, however, are then input for a subsequent synthesis step. Ultimately, a controller for the whole specification must be synthesized, which is not the case in our approach. While their approach may be more flexibly applicable, our approach can often more drastically reduce the time required by the synthesis.

Maoz and Sa'ar recently proposed a technique for synthesizing controllers from LSC specifications with environment assumptions [16], but they do not address the decomposition of the synthesis problem. Their approach also differs from ours in the way that assumptions are formulated. They propose to model environment assumptions by specially labeled environment messages in LSCs. We instead propose to model assumptions by specially labeled MSDs. Only this makes it possible to model the same property as requirements in one specifications and assumptions in another, which is the key to our technique.

Chatterjee and Henzinger also present a compositional assume-guarantee synthesis approach from specifications in temporal logic [4]. They, however, regard a different problem: translated into our terminology, they regard the problem of synthesizing controllers for two system objects that interact with an environment

and have local, possibly interdependent specifications. The goal is to synthesize two controllers that fulfill each system object's local specification without violating the specification of the other system object. This process, called *co-synthesis*, does not aim at being more efficient than synthesizing a global controller—in general the problem is even more complex. They, however, sketch an abstraction approach to make the co-synthesis more efficient.

Nejati et al. present a compositional approach for synthesizing sequential compositions of *features*. Features are units of functionality that are modeled with state machines, to fulfill certain requirements [17]. They, however, are only considering to find a viable composition of features and do not consider the synthesis of state machines themselves.

Krüger proposes a mapping from (High-Level) Message Sequence Charts to assume-guarantee specifications of components [12]. The scenario language regarded by Krüger, however, does not allow for flexible overlappings of scenarios as it is allowed for LSCs or MSDs. So the resulting synthesis problem is more simple than the MSD/LSC synthesis problem that we consider.

7 Conclusion and Outlook

We presented a novel compositional synthesis technique for scenario-based specifications, which makes use of the assume-guarantee paradigm. The technique allows engineers to decompose the problem of synthesizing a controller for an MSD specification into two synthesis problems that can be solved independently from each other. This can significantly reduce the overall computation time for synthesizing the controllers. We provided a soundness proof and some evaluation results that document the benefit of our technique.

A limitation of our technique is that we currently allow only for one controller to make assumptions about the other. The reasons for this lies in the nature of liveness properties: a violation of a liveness property cannot be pinpointed to a specific point of the run at which it is violated. Therefore, if both controllers violate some assumptions which are liveness properties, it is not clear which one violated its assumption first. If there were cyclic assumptions and guarantees concerning liveness properties, each controller could blame the violation on the other. Therefore, no component would need to guarantee anything.

There are different ways of dealing with this problem. One idea is applying a concept for composing controllers proposed in [11]. This concept relies on explicit dependency graphs between the involved assume-guarantee properties of a components, which need to stay acyclic when combining components. Another idea would be to apply the compositional synthesis technique of Chatterjee and Henzinger [4] (see also Sect. 6), if by using the described abstraction techniques the co-synthesis problem can be sufficiently simplified.

References

1. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–235 (1994)

2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) Proc. 19th Int. Conf. on Computer Aided Verification (CAV'07). LNCS, vol. 4590, pp. 121–125. Springer (July 2007)
3. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) Proc. 16th Int. Conf. on Concurrency Theory (CONCUR'05). LNCS, vol. 3653, pp. 66–80. Springer, San Francisco, CA, USA (August 2005)
4. Chatterjee, K., Henzinger, T.A.: Assume-Guarantee Synthesis. In: Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07). pp. 261–275. Springer (2007)
5. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
6. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. In: Formal Methods in System Design. vol. 19, pp. 45–80. Kluwer Academic Publishers (2001)
7. Greenyer, J.: Scenario-based Design of Mechatronic Systems. Ph.D. thesis, University of Paderborn (October 2011)
8. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software and Systems Modeling (SoSyM) 7(2), 237–252 (May 2008)
9. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (August 2003)
10. Harel, D., Marelly, R.: Playing with time: On the specification and execution of time-enriched LSCs. In: Proc 10th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. pp. 193–202 (2002)
11. Kindler, E.: Modularer Entwurf verteilter Systeme mit Petrinetzen, Edition Versal, vol. 1. Bertz Verlag (Dec 1995), dissertation, Technische Universität München
12. Krüger, I.: Distributed System Design with Message Sequence Charts. Ph.D. thesis, Technische Universität München, Institut für Informatik (2000)
13. Kugler, H., Segall, I.: Compositional synthesis of reactive systems from live sequence chart specifications. In: Proc. 15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09). LNCS, vol. 5505, pp. 77–91. Springer (2009)
14. Lewerentz, C., Lindner, T.: KORSO: Methods, Languages, and Tools for the Construction of Correct Software, LNCS, vol. 1009, chap. Case study "production cell": A comparative study in formal specification and verification, pp. 388–416. Springer (2006)
15. Maoz, S., Harel, D.: From multi-modal scenarios to code: Compiling LSCs into AspectJ. In: Proc. Int. 14th Symp. on Foundations of Software Engineering (FSE'05). pp. 219–230. ACM (2006)
16. Maoz, S., Sa'ar, Y.: Assume-guarantee scenarios: semantics and synthesis. In: Proc. 15th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS'12). LNCS, vol. 7590, pp. 335–351. Springerg (2012)
17. Nejati, S., Sabetzadeh, M., Chechik, M., Uchitel, S., Zave, P.: Towards compositional synthesis of evolving systems. In: Harrold, M.J., Murphy, G.C. (eds.) Proc. 16th Int. Symp. on Foundations of Software Engineering. pp. 285–296. ACM (2008)
18. Stark, E.W.: A proof technique for rely/guarantee properties. In: Foundations of Software Technology and Theoretical Computer Science, LNCS, vol. 206, pp. 369–391. Springer (1985)