Synthesizing Safe Dynamic Updates from Evolving Specifications

Technical Report n. 2013.2 - DEIB Politecnico di Milano

Joel Greenyer[1], Valerio Panzica La Manna[1], Christian Brenner[2], and Carlo Ghezzi[1]

[1] *Dependable Evolvable Pervasive Software Engineering (DEEPSE) Group, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano, Italy*

[2] *Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Zukunftsmeile 1, 33102 Paderborn, Germany*

January 2013

Abstract

Many software-intensive systems are expected to run continuously while their environments change or their requirements evolve. The vision is therefore to build systems that update dynamically during run-time and adapt their behavior without disruption. Especially in critical applications, we must ensure that dynamic updates are safe and are performed as soon as possible. In recent work, we formalized a criterion of when a dynamic update of a current finite-state controller is correct with respect to a specification change—providing a new, specification-oriented view of dynamic updates. We also proposed a systematic approach for automatically synthesizing dynamically updating controllers from changes in scenario-based specifications. In this paper, we extend our previous work in two directions. We present a more efficient technique for synthesizing dynamically updating controllers incrementally and report on its implementation in a novel, model-based synthesis tool.

This research is funded by the European Commission, Programme IDEAS-ERC, Project 227977 SMScom. Christian Brenner is supported by the International Graduate School Dynamic Intelligent Systems.

1 Introduction

Advanced technical systems in transportation or production, but also information systems in commerce and healthcare, often have to operate continuously while their environments change or their requirements evolve. It is often expensive and impractical to shut down these systems in order to perform software updates, so the goal is to update the software during run-time. But especially

in critical applications, we must ensure that run-time updates are *safe*, are performed *without disruption*, and we desire that the system adapts to specification changes *as soon as possible*.

Techniques to develop systems that update their software to a new version during run-time, called *dynamically updating systems* or *dynamic software updates (DSU)*, have been intensively studied in the past [8, 22, 31, 26, 16, 17, 7, 3, 4, 21, 6]. However, the existing approaches do not investigate the relationship between the evolving specification of the system and the updating software.

Most existing approaches just consider that the system or its parts update in a state where they are not currently involved in any interaction [22, 31, 26], or that a component can be replaced if it still provides the services of the old [7]. In programming languages, there are similar approaches that require procedures to be inactive during the update [24, 16], or that the current program state is also a state of the new program version [17]. But these approaches do not consider whether the dynamic updates are correct with respect to specification changes.

There also has been intensive research on *dynamically adaptive* systems, which are systems that dynamically change their behavior during run-time [14, 7, 33, 2, 12]. Languages have been proposed that typically allow for designing software that can re-configure between a fixed set of configurations at predefined update points. Also, techniques exist for checking adaptation-specific properties of the adaptive software. But again, it has not yet been studied when a system adapts correctly with respect to changing requirements or environment properties.

We consider a setting where system and environment components interact via messages, and where the system components are controlled by one global finite-state controller. For simplicity, we consider only synchronous messages, where the sending and receiving of the message is a single event, but our approach can also be extended to support asynchronous messages. Extending our approach to distributed controllers, where each system component is controlled by its own controller, remains an outlook.

The controller implements a specification, which consists of *requirements* and *environment assumptions*, as in many assume-guarantee specification approaches [30, 29, 1]. The requirements describe allowed infinite sequences of environment and system events and thereby specify how the system must react to events in the environment. Similarly, the environment assumptions specify what sequences of environment events are possible to occur, or also how the environment in turn reacts to events in the system. A system must satisfy the requirements if the environment satisfies the environment assumptions.

Given a current controller that implements a current specification, and a change in the requirements and/or environment assumptions, we want to know:

1. *In which states* of the running system is it safe to disregard the current obligations of the system and update the behavior to satisfy the new, changed specification?

2. If in such a state, how *must the system adapt* to satisfy the new specification?

3. Based on the current controller and the specification change, how can we *derive a controller* that dynamically updates to the new behavior as soon as possible?

In recent work, we proposed a first definition of *updatable states* and *correct dynamic updates* of a current finite-state controller with respect to a specification change—providing a new, specification-oriented view on dynamic updates [13]. Moreover, we presented an approach for automatically synthesizing dynamically updating controllers from changes in Modal Sequence Diagram (MSD) specifications. MSDs [19] allow engineers to intuitively and visually specify different sequences of events that may, must, or must not happen in the system.

This paper extends our previous work [13] by presenting a technique for synthesizing dynamically updating controllers *incrementally* from changes in scenario-based specifications and the given current controller. This has the following advantages compared to the previous approach: (1) being incremental, the algorithm is significantly more efficient, (2) we simplify the overall approach of synthesizing a dynamically updating controller, and (3) it improves the practical applicability of the approach since the resulting controller has a smaller number of states and is deterministic, which is typically desired for final implementations. Moreover, we implemented this approach in ScenarioTools, a novel, Eclipse-based synthesis and simulation tool suite for MSD specifications[1].

In the scope of this paper, we consider that specification changes are specified by an engineer, but our vision is that, in the future, specification changes may also be derived by the system at run-time, e.g., from high-level goals or user input [11, 32]. Likewise, changes in environment assumptions may also be computed by the system as it monitors its environment. Thus, we consider our approach a first step towards developing safe and automatic self-adaptation mechanisms that are driven by changes in requirements and environment properties.

The paper is structured as follows. Based on intuitive examples in Sect. 2, we give a formal definition of updatable states and correct updates in Sect. 3. We then introduce evolving scenario-based specifications in Sect. 4. We describe a systematic approach for synthesizing dynamically updating controllers in Sect. 5, and explain the optimized, incremental synthesis approach in Sect. 6. In Sect. 7 we describe our tool implementation. Last, we discuss related work in Sect. 8 and conclude in Sect. 9. We also include an appendix with more details on the algorithms used in our approach.

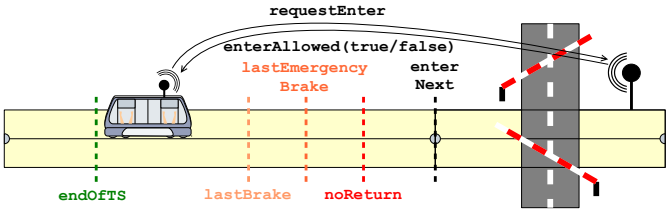
2 Requirements Evolution Examples

As a running example, we consider an evolving specification of the RailCab system[2], a system developed at the University of Paderborn where autonomous vehicles, called *RailCabs*, transport passengers and goods on demand.

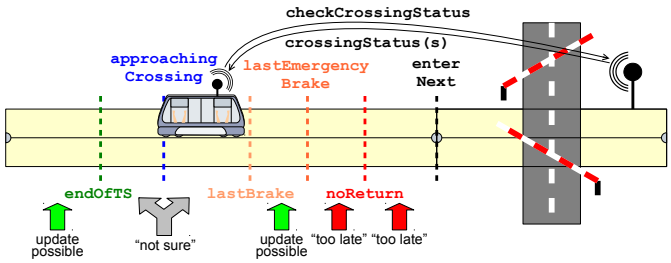
Let us consider the simplified requirements of what shall happen when a RailCab approaches a crossing. Figure 1(a) shows a RailCab approaching the end of its current track section to enter a crossing. There are different environment events it receives (observed by sensors or computed by low-level components) in a certain order as it approaches the crossing. First, the RailCab detects that it approaches the end of the current track section (`endOfTS`). Then it must send a request to the crossing control for the permission to enter

[1] The ScenarioTools tool suite, `http://scenariotools.org`

[2] "Neue Bahntechnik Paderborn", `http://www-nbp.upb.de`



(a) check crossing status after `endOfTS`, before `lastBrake`



(b) check crossing status after `approachingCrossing`, before `lastEmergencyBrake`

Figure 1: A RailCab approaches a crossing

(`requestEnter`), which must reply whether entering the crossing is allowed or not (`enterAllowed(true/false)`).

This reply must be sent before `lastBrake` occurs. Between `lastBrake` and `lastEmergencyBrake` the RailCab can avoid entering the crossing by performing a normal braking procedure. Between `lastEmergencyBrake` and `noReturn`, the RailCab can avoid entering the crossing only by applying the emergency brakes. After `noReturn`, the point of no return, the RailCab can no longer avoid entering the crossing (`enterNext`).

Now suppose that the requirements change, because it was observed that a power outage may lead to a situation where the RailCab enters a crossing while the crossing control could not shut the barriers, thus increasing the risk of accidents. It is now additionally required that the RailCab must check the crossing's operational status by sending the message `checkCrossingStatus` to the crossing control, which must in turn reply with its status via the message `crossingStatus(s:Status)`. This interaction shall take place not immediately, but some time after `endOfTS`, and before `lastEmergencyBrake`. For this purpose, we suppose that another signal was installed on the track, called `approachingCrossing`, which shall trigger this interaction. The RailCab will pass this point after `endOfTS` and before `lastBrake`. Figure 1(b) illustrates the additional requirement and the additional environment event.

But when is it possible to change the behavior so that the new requirements can be satisfied? We would like to update to the new behavior as soon as possible, even if a RailCab is already approaching a crossing, to avoid deadly accidents.

We suppose that one valid option would be to perform an offline update, i.e., to shut down the system, to update its controller to a new version that satisfies the new specification, and to restart the system, with the new controller in its initial state. Intuitively, performing a dynamic update when the controller

is in its initial state would lead to an equivalent behavior, in terms of the sequences of the above-mentioned events. Thus, it would be possible in this case to update the system before the event `endOfTS` occurs. Likewise, we could also wait with the update until after having entered the crossing, so that we satisfy the new requirements the next time we approach a crossing. Here we assume the simplified case where the RailCab is not involved in any other interactions.

There are even more states where we can update and still achieve a behavior that is equivalent to an offline update. For example, after `endOfTS` has already occurred, it is still possible to perform an update, because nothing has happened yet that is forbidden according to the new specification. In other words, it is still possible to complete the sequence of events that took place since the controller last visited the initial state to a run that satisfies the new specification. It would even be possible to complete the run to satisfy the new requirements after `requestEnter` or `enterAllowed` was sent, assuming that this interaction takes place immediately after `endOfTS`.

However, we can assume that an implementation of the old specification does not monitor or remember the newly introduced environment event `approachingCrossing`. Thus, after `endOfTS`, the controller will not know whether `approachingCrossing` has already occurred or not. If we update the system assuming that this event did not yet occur, whereas it did in fact occur, we may miss the occurrence of the event and never trigger the RailCab to request the crossing's operational status. This would violate the new requirements and could have devastating consequences, especially if we imagine an example where some existing safety-mechanism is replaced by a new one. If we update the system assuming that `approachingCrossing` has already occurred whereas in reality it didn't, our updated software would check the crossing's operational status before `approachingCrossing` has actually occurred. Here this would not lead to a dangerous situation, but it is invalid with respect to the new specification.

After `lastBrake` occurred, and because we know that `approachingCrossing` must have happened before that, it would again be possible to complete the run to satisfy the new requirements. After `lastEmergencyBrake` occurred, it is however too late, because the RailCab should have checked the crossing status before.

This example shows that a running system, depending on its state and what we assume happened in the environment, may or may not be updatable to a changed specification.

3 Correct Dynamic Updates

We formally define updatable states and correct updates in Sect. 3.2. Before that, we give preliminary definitions in Sect. 3.1.

3.1 Object Systems, Controllers, Runs, Specifications

We consider systems of *objects* that exchange messages as defined by Harel and Marelly [20]. We only consider synchronous messages.

Definition 1 (Object system, message event, alphabet, run) *An* object system *consists of a set of objects O that exchange* messages. *A message has*

a name and a sending and receiving object (i.e., it is assumed to be point-to-point). The sending and receiving of a message is a single event*, also called a* message event*. The* alphabet Σ *is the set of different message events that can occur in an object system. An infinite sequence of message events $\pi \in \Sigma^\omega$ is called a* run *of the system.*

The objects in the system are controlled by a *controller*. A controller can control one or more objects, but one object can only be controlled by one controller.

Definition 2 (Controller, trace language) *A* controller *is a finite state machine without final states: a finite state machine is a quadruple (Σ, Q, q_0, T), where $Q = \{q_0, \ldots, q_n\}$ is a finite set of states, q_0 is the start state (or initial state) and $T \subseteq Q \times \Sigma \times Q$ is a transition relation. For a controller c, $L(c) \subseteq \Sigma^\omega$ is the* trace language *of c. A run $\pi = (m_0, m_1, \ldots)$ is an element of $L(c)$ iff there exists a sequence of states starting from the start state of the controller $(q_0, q_1, \ldots) \in Q^\omega$ such that $\forall i \geq 0 : (q_i, m_i, q_{i+1}) \in T$.*

A controller can also consist of the parallel composition of two controllers that control disjoint subsets of objects. The composed controllers synchronize on message events sent between an object in one set to an object in the other set.

Definition 3 (Parallel composition) *Let $c_1 = (\Sigma_1, Q_1, q_{0_1}, T_1)$ and $c_2 = (\Sigma_2, Q_2, q_{0_2}, T_2)$ be two controllers for disjoint sets of objects. Furthermore, let Σ_1 only be such events where the sending or receiving object is controlled by c_1 and let Σ_2 only be such events where the sending or receiving object is controlled by c_2. The* parallel composition *of c_1 and c_2, written $c_1||c_2$, is equivalent to a controller $(Q_1 \times Q_2, (s_{0_1}, s_{0_2}), \Sigma_1 \cup \Sigma_2, T_1||T_2)$ where $Q_1 \times Q_2$ is the set of all possible tuples of Q_1 and Q_2, and $T_1||T_2$ is a transition relation defined as follows:*

1. *$((s_1, s_2), m, (s'_1, s_2)) \in T_1||T_2$ if there is a transition for the event m in controller c_1, $(s_1, m, s'_1) \in T_1$, and m is not sent or received by any object controlled by c_2, $m \notin \Sigma_2$.*

2. *$((s_1, s_2), m, (s_1, s'_2)) \in T_1||T_2$ if there is a transition for the event m in controller c_2, $(s_2, m, s'_2) \in T_2$, and m is not sent or received by any object controlled by c_1, $m \notin \Sigma_1$.*

3. *$((s_1, s_2), m, (s'_1, s'_2)) \in T_1||T_2$ if there is a transition for the event m in both controllers, $(s_1, m, s'_1) \in T_1$ and $(s_2, m, s'_2) \in T_2$.*

Last, we define a specification and when a system satisfies and implements it.

Definition 4 (Specification, satisfying a specification) *A* specification S *is a tuple (A, R) with the* assumptions A *and the* requirements R *being sets of runs. A run π* satisfies *the specification S, written $\pi \models S$ iff $\pi \in A \Rightarrow \pi \in R$, i.e., if the run is in the assumptions, it must also be in the requirements. A controller c* satisfies *S, written $c \models S$, iff each run in $L(c)$ satisfies S.*

Definition 5 (System and environment objects) *The objects of the system can be either controllable* system objects *or uncontrollable* environment objects. *For a controller of the environment objects, we require that in every state there are outgoing transitions by which it can receive all events sent from system objects to environment objects. For a controller of the system objects we require that it infinitely often is in a state with outgoing transitions by which it can receive any event sent from environment objects to system objects.*

Intuitively, this means that the environment can never block any event occurring in the system and that the system can perform any finite number of steps, but must eventually listen for the next environment event.

Definition 6 (Implementation) *A controller c for all the system objects* implements *or* realizes *S iff c composed with every possible controller e for the environment objects satisfies S, more formally $\forall e, e||c \models S$.*

In the scope of this paper, we consider a setting where all system objects are controlled by a single controller, also called the *global* controller.

3.2 Histories and Updatability

We now return to the problem of understanding when a dynamic update can be safely performed. In Sect. 2, we intuitively argued that a dynamic update is safe if it leads to a behavior that is equivalent to an offline update, which involves shutting down the system and restarting the system with a new controller in its initial state. This equivalence, in the following, is the basis for our definition of *correct dynamic updates*.

Without considering the details on how to shut down a system, we assume that a system controlled by a controller c that implements a given specification S can always be "safely" shut down. By "safely" we mean that shutting down and restarting a system (without an update) will not violate the specification S. We can thus imply that for every run of c, there exists an equivalent run where, if c is visiting its initial state, the system is instead shut down and restarted in the initial state of c.

A dynamic update of a current controller c that implements a specification S to a changed specification S' would thus be equivalent to an offline update if it is equivalent to a run of a system controlled by c that eventually reaches its initial state, and is then replaced by a new controller c' in its initial state, where c' implements S'.

This notion of dynamic update fits the case of ever-running systems that have a periodic behavior whereby they infinitely often visit their initial state. This case is typical of many systems; generalizations of this notion that fit other kinds of systems will be subject of future work.

If the current controller is in its initial state, a correct dynamic update is thus straightforward. But there are more, later states in which we can yet modify the controller to achieve the same behavior. We call these states *updatable states*.

Intuitively, a state of a controller c is an updatable state if the sequence of events that led to it from the initial state can be completed to a run that satisfies the changed specification. However, if the system is currently in a state q_{cur}, we have to assume that different sequences of events may have occurred in the past. This could even be sequences of events for which the controller passed the initial

state multiple times. But we only consider the possible sequences of events since the latest visit of the initial state. We call these possible sequences of events the possible *recent histories* of a state q_{cur}. For a state to be updatable, we require that it must be possible to continue executing the system so that *every* possible recent history can be completed to a run that satisfies the changed specification S'.

This condition alone, however, is not sufficient. We also require that there must not be any confusion on how to continue executing the system. This means that, it must be possible to continue executing the system so that *one* continuation completes *every* possible recent history to a run that satisfies S'.

By considering all the possible recent histories, our approach does not rely on any additional logging mechanism to remember the exact sequence of past events leading to q_{cur}. The problem of a logging mechanism is that the required size of the log depends on the new specification and thus cannot be known a priori, so a logging mechanism would not solve the problem in general.

To determine the possible recent histories of a state in the controller, we must also include what we assume has possibly happened in the environment; the system controller may not capture everything that happens in the environment. The possible environment behavior is described in the environment assumptions—the question is just, in the case where the environment assumptions have changed, whether we think that the environment *will yet change in the future* or we assume that the environment *has changed already.*

In the latter case, we should consider the changed environment assumptions in a changed specification S'. Very often changes in the environment assumptions reflect *new insights* in how the environment is *already behaving right now*, which have just not been captured thus far. The event `approachingCrossing` in the RailCab example (see Fig. 1(b)) could be a signal along the track section that is already detected by the RailCab's sensors, but was thus far ignored by the controller. Here the changed environment assumptions should be considered to determine the possible recent histories.

If, however, changes in the environment assumptions describe changes in the environment behavior *that will take effect during or after the update*, then the old environment assumptions must be considered to determine the possible recent histories.

To stress this difference, we assume the former case in the following and determine the possible recent histories based on an environment e' that satisfies the changed environment assumptions A'.

Definition 7 (histories, recent histories) *We consider the composition of c with every possible controller for the environment e' that satisfies the assumptions of S' such that $L(e'||c) \subseteq A'$. The* histories $\Pi_{past}(c, q_{c_{cur}})$ *are the paths from the start state of $e'||c$ to a state where c is in $q_{c_{cur}}$,* $\Pi_{past}(c, q_{c_{cur}}) = \{(m_1, .., m_n) \in \Sigma^* \; s.t. \; \exists (q_0, .., q_n) \in (Q_{e'} \times Q_c)^*, q_0 = (q_{e'_0}, q_{c_0}), q_n = (q_{e'_n}, q_{c_{cur}})$ *and* $\forall i \in \{0, .., n\} : (q_i, m_i, q_{i+1}) \in T_{e'}||T_c\}$. *The* recent histories $\Pi^{<}_{past}(c, q_{c_{cur}})$ *are such histories where the start state is not visited a second time, i.e.,* $\forall i > 0 : q_0 \neq q_i$.

Based on the possible recent histories, we define updatable states of a controller with respect to a changed specification as follows.

Definition 8 (Updatable state, correct update) *A state q_{cur} of a system controller c is* updatable *to a specification S' iff there exists a controller c' that implements S' and where the composition with any possible environment controller e' has a trace language $L(e'||c')$ where*

1. *for every recent history $\pi^{<}_{past} \in \Pi^{<}_{past}(c, q_{c_{cur}})$ there must be a run $\pi \in L(e'||c')$ where $\pi^{<}_{past}$ is a prefix of π. In other words, there must exist a continuation of any possible recent history, which we call π_{future}, that concatenated with $\pi^{<}_{past}$ forms π, $\exists \pi_{future} \in \Sigma^{\omega} : \pi = \pi^{<}_{past} \cdot \pi_{future}$.*
2. *If π_{future} is a continuation of some possible recent history, it must also be a continuation of any other possible recent history, $\forall \pi^{<1}_{past}, \pi^{<2}_{past} \in \Pi^{<}_{past}(c, q_{c_{cur}}), \pi^{<1}_{past} \neq \pi^{<2}_{past} : \pi^{<1}_{past} \cdot \pi_{future} \in L(e'||c') \Rightarrow \pi^{<2}_{past} \cdot \pi_{future} \in L(e'||c')$*

A system with controller c in an updatable state q_{update} performs a correct update *to satisfy the changed specification S' iff the sequence of events that occurred since c was in the initial state for the last time will be completed to a run that satisfies S'.*

According to this definition, the controller of the RailCab would be updatable to the changed specification as described in Sect. 2 in a state where `endOfTS` did not yet occur, see Fig. 1(b). This is obvious, since this state corresponds to the controller's initial state, and thus the recent histories are empty. So, if the changed specification S' is consistent, there exists a controller c' implementing S' that can continue the recent histories as defined above.

After `endOfTS` and before `lastBrake` occurred, the controller is in a state that is not updatable. This is because in this state the controller does not monitor or remember that the environment event `approachingCrossing` has occurred. Therefore, there are two different recent histories in this state, one where the event occurred, and one where it did not occur. For both of these histories, possible continuations exist, but the possible continuation of one is not a possible continuation of the other, i.e., there would be a confusion on how to continue executing the system.

However, after `lastBrake` occurred, we are sure, due to the environment assumptions, that `approachingCrossing` has already occurred. In the state before `lastEmergencyBrake`, all the recent histories can still be completed to a run that satisfies the changed specification. In a state after `lastEmergencyBrake`, the recent histories cannot be completed to satisfy the changed specification, because in the recent histories the messages `checkCrossingStatus` and `crossingStatus` were not sent, which is a violation of the changed specification, no matter how the run is completed.

Our definition implies another interesting condition for updatable states.

Lemma 1 (Target state) *If a state q of a system controller c is updatable to a specification S', this implies that there exists a controller c' with* exactly one *state q' for which holds: $\forall \pi^{<}_{past} \in \Pi^{<}_{past}(c, q)\ \exists \pi_{future} \in \Pi_{future}(c', q') : \pi^{<}_{past} \cdot \pi_{future} \models S'$, where $\Pi^{<}_{past}(c, q)$ is the set of recent histories of q as defined above and $\Pi_{future}(c', q')$ is the set of infinite runs of $e'||c'$ starting in a state were c' is in q'. A state for which this condition holds in a controller implementing the changed specification S' is called the* target state *of a state q in the current controller.*

This means, intuitively, that if a state is updatable to a new specification S', there must exist a controller with exactly one state that we can "update to", i.e., when reaching the updatable state, we can stop executing the current controller and resume executing this new controller in the corresponding target state. This will then result in a correct update behavior.

The fact that there exists a controller with at least one such target state follows directly from the definition of updatable states (Def. 8). That there must exist a controller where there is only one such target state can be shown simply as follows. Consider a controller with more than one target state. Then, for each transition sequence starting from one of these states, there must also be a corresponding transition sequence, labeled with the same events, starting from any other of these states (this follows from Def. 8.2). This implies, however, that there exists an equivalent controller, accepting the same language, but where these target states are reduced to one state.

4 Evolving Scenario-based Specifications

We propose an approach where the specification of the system is given in the form of Modal Sequence Diagrams (MSDs) [19], a formal interpretation of UML sequence diagrams based on the concepts of Live Sequence Charts (LSCs) [9, 20]. MSDs generalize some concepts of LSCs (see [19] for details). This section explains the syntax and semantics of MSDs and shows an example of how an MSD specification may evolve.

4.1 MSD Specifications

An MSD specification consists of a description of the object system and a set of MSDs where each lifeline represents exactly one object in the system. Figure 2 shows the MSD specification Drive onto crossing. The object system is described in the form of a UML collaboration diagram. Here we consider specifications where the MSDs can be either *requirement MSDs* or *assumption MSDs* [15]. The latter are annotated with the ≪EnvironmentAssumption≫ in their name label.

Both kinds of MSDs are *universal MSDs*, which describe properties that are required (or assumed) for every run of the system and its environment. The requirement MSD RequestEnterAtEndOfTrackSection in Fig. 2 for example formalizes the properties described informally in Sect. 2. It says that when the RailCab detects that it approaches the end of the track section (`endOfTS`), it must request the permission to enter the crossing (`requestEnter`). Then the crossing control must reply, stating whether entering the crossing is allowed or not (`enterAllowed`). This must happen before the RailCab passes the point where it is guaranteed for the last time that by braking it will stop before entering the switch (`lastBrake`).

The assumption MSD PassingPointsOnTrack says that when the RailCab detects that it approaches the end of the track section (`endOfTS`), it will also eventually pass the point of the last safe brake (`lastBrake`), and last emergency brake (`lastEmergencyBrake`), the point of no return (`noReturn`), and will then finally enter the crossing (`enterNext`). For simplicity, we assume that the RailCab does not brake or reverse.

4.2 Message Temperature, Execution Mode, and Parameters

The messages in a universal MSD have a *temperature* and an *execution kind*. The temperature can be either *hot* or *cold*. The execution kind can be either *monitored* or *executed*. In Fig. 2 and 3, we indicate the temperature and execution kind of the messages by attached labels, e.g. (c/m) for cold monitored messages and (h/e) for hot executed messages. Hot/cold messages are also shown as red/blue arrows. Executed messages have a solid line, monitored ones have a dashed line.

The semantics of these messages is as follows. We consider that an MSD always has one first message. When an event occurs in the system that can be *unified* with the first message in an MSD, an *active copy* of the MSD or *active MSD* is created. An event can be unified with a message in an MSD if the event name equals the message name and the sending and the receiving object are represented by the sending resp. receiving lifeline of the message. There can be multiple active MSDs at a time.

As further events occur that can be unified with the subsequent messages in the diagram, the active MSD progresses. This progress is captured by the *cut*, which marks for every lifeline the locations of the messages that were unified with the message events. If the cut reaches the end of an active MSD, the active copy is terminated.

If the cut is immediately before a message on its sending and receiving lifeline, this message is *enabled*. If a hot message is enabled, the cut is *hot*, otherwise the cut is *cold*. If a message is enabled with an execution kind set to execute, the cut is *executed*, otherwise the cut is *monitored*. An enabled executed message is also called an *active message* or *active event*.

If the cut is hot, it is not allowed for events to occur that can be unified with another message in the MSD that is not enabled, otherwise this is called a *safety violation*. If the cut is cold, such messages are allowed to occur, but then the active MSD is terminated. This is called a *cold violation*. Messages that cannot be unified with any message in the MSD are ignored. Moreover, it must not happen that from some point on there is forever always at least one active MSD with an executed cut. This case is called a *liveness violation*. The dashed lines in Fig. 2 show the reachable cuts. Labels indicate whether a cut is hot or cold and monitored or executed.

Messages can also have parameters of certain types. A message event then carries according values for each parameter. We only consider messages with at most one parameter, which poses no fundamental restriction. Messages in the MSDs can either specify a concrete value for messages or they can be *symbolic* and specify no parameter value. The MSD RequestEnterAtEndOfTrackSection in Fig. 2 for example shows the message `enterAllowed`, which has a Boolean parameter representing the choice to allow or deny the RailCab to enter. In this MSD, the message is symbolic, we write "t/f" to express that the message does not specify a particular value for the parameter. A message event is *parameter unifiable* with a diagram message iff the diagram message is symbolic or specifies the same parameter value as carried by the message event. If a parametrized message is enabled, the cut progresses if the event is parameter unifiable with the diagram message. It is a cold/safety violation (depending on the temperature of the cut and in addition to the previous notion of cold/safety violations) if the

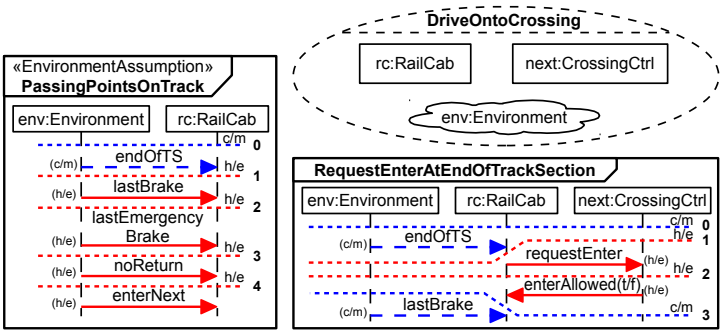


Figure 2: The example MSD specification (S)

event is unifiable, but not parameter unifiable with an enabled diagram message.

4.3 Play-out and Controller Synthesis

For LSC and MSD specifications there exists an algorithm for directly *executing* the scenarios, which is called *play-out* [20, 27]. The basic idea is the following: As environment events occur, MSDs are activated and reach a state where in one or more active MSDs, there is an active event, i.e., an executed enabled message. Then, the play-out algorithm (non-deterministically) chooses one of the active messages and executes it. It repeats this process until all active MSDs are terminated or reach a non-executed cut, where it waits for the next environment event to occur, etc. This algorithm is useful for engineers to simulate their specification and understand the interplay of the scenarios.

Also controllers that implement an MSD specification can be synthesized automatically [18, 5, 23, 15]. The synthesis problem can be seen as the problem of finding a winning strategy in an infinite two-player game where the system always tries to satisfy the specification and the environment will try everything to violate the specification [5, 23]. We also implemented synthesis algorithms in a ScenarioTools. First, we can synthesize a *maximal controller*, which contains all reactions by which a controller can satisfy a specification. Second, we can synthesize a controller that contains only one solution for how the system can satisfy a specification. The latter algorithm we explain in more detail in Sect. 6.

4.4 Evolving MSD Specifications

MSD specifications can evolve by adding or removing MSDs to or from the specification. If a requirement MSD is added to the specification, it means that an additional requirement must be satisfied; if it is instead removed, some property need no longer be satisfied. If an assumption MSD is added to the specification, it means that an additional property can be assumed about the environment; if it is instead removed, it means that some assumption about the environment is no longer valid.

As an example, we consider a specification change as explained informally in Sect. 2. Fig. 3 shows the change in the MSD specification: (1) We replace the assumption MSD PassingPointsOnTrack by one that also contains the en-

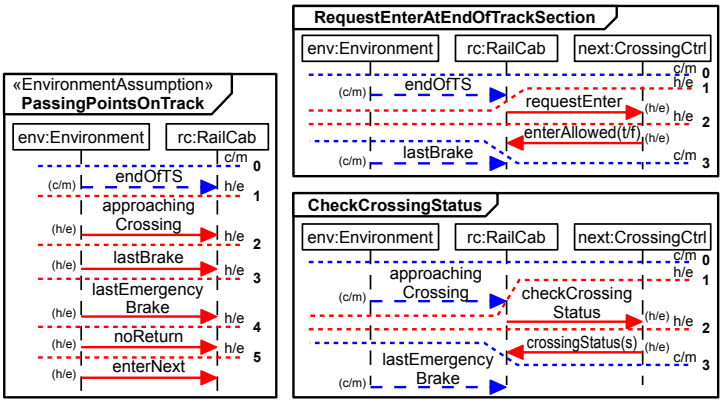


Figure 3: The changed MSD specification (S')

vironment event `approachingCrossing`. (2) We add the requirement MSD CheckCrossingStatus that describes, similar to the MSD RequestEnterAtEndOfTrackSection, that the RailCab must check the operational status of the crossing after `approachingCrossing` and before `lastEmergencyBrake` occurred.

5 Synthesizing Dynamically Updating Controllers

In this section, we describe a systematic, yet not efficient, approach to synthesize a *dynamically updating controller*. The explanation is supported by an example presented in Sect. 5.2. We justify the correctness of the systematic approach in Sect. 5.3. In Sect. 6, we present our more efficient, incremental approach.

5.1 A systematic approach

Given a current controller c, implementing the current specification S, and a changed specification S', we synthesize a *dynamically updating controller*. This controller behaves as the current controller and, as soon as an updatable state is reached, dynamically updates to the new behavior. This is illustrated in Fig. 4.

The dynamically updating controller, shown on the bottom-right of Fig. 4, basically consists of the current controller c and the structure of a new controller c' that implements S'. In updatable states of c, we add *update transitions* to particular states in the c'-part, where the execution of the system is continued according to the changed specification. All other outgoing transitions of updatable states in the c-part are removed. Added states and transitions are labelled with "++", removed transitions are crossed-out. This adding and removing of controller parts, while maintaining the current runtime state, can also be understood as an installation procedure to "patch" a current, running controller to a dynamically updating one, but we don't consider this procedure in more technical detail.

The synthesis of the dynamically updating controller is achieved in the following steps:

1. We require a model that contains all the possible recent histories of all the states in c. We thus create a model of the overall behavior of the

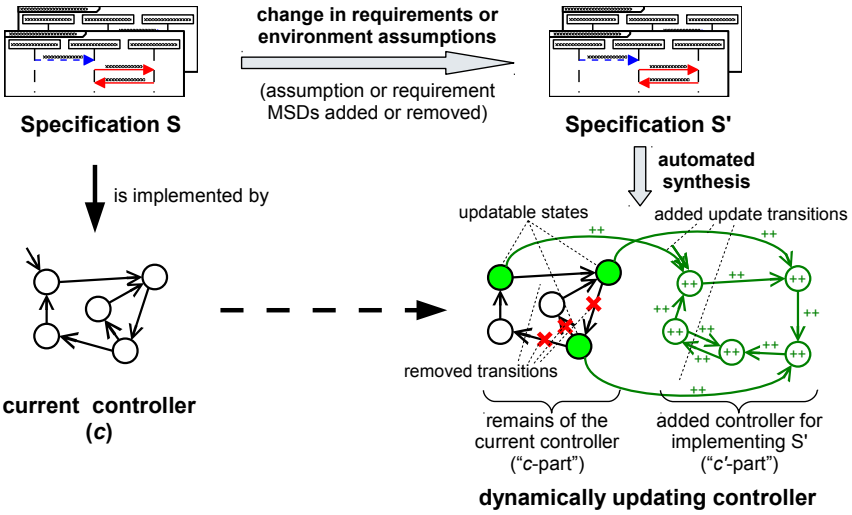


Figure 4: The approach for synthesizing a dynamically updating controller from a specification change and the current implementation of the system

system in any environment that behaves according to the new environment assumptions A'. This is done by creating the parallel composition $e'||c$, where e' is the maximal controller for A'[3].

2. We need a model of the overall behavior of the new system. We therefore synthesize the maximal controller c', which implements the new specification S'. As above, we then compute the parallel composition $e'||c'$, which results in a controller that contains all the possible recent histories of every state in c'.

3. We establish the *history relation* between states in $e'||c$ and $e'||c'$. This relation maps a state in $e'||c$ to a state in $e'||c'$ iff every recent history of the first is also a recent history of the second.

4. From the history relation between $e'||c$ to a state in $e'||c'$, we construct the dynamically updating controller as follows:

   (a) We structurally combine c with c'.

   (b) We "transform" the history relation into update transitions if there is a unique target state for the update: We create a transition from a state q_1 in c to a state q'_1 in c' iff the history relation contains at least one mapping of a state $(q_{e'}, q_1)$ in $e'||c$ to a state $(q'_{e'}, q'_1)$ in $e'||c'$ (for some states $q_{e'}$ and $q'_{e'}$ of e'), but there is no other mapping from $(q_{e'}, q_1)$ in $e'||c$ to a state $(q''_{e'}, q'_2)$ in $e'||c'$ where $q'_2 \neq q'_1$ for some state $q''_{e'}$ of e'.

   (c) From states where we added update transitions, i.e., the updatable states, we remove all other outgoing transitions.

[3]We can synthesize a controller (maximal or not) from the assumptions by interpreting the assumptions as requirements in a new specification.

(d) Start state of the resulting controller is the state corresponding to the start state of c'.

The algorithm for computing the history relation between the controllers $e'||c$ and $e'||c'$ works as follows. First, we compute the pair of states of the controllers $e'||c$ and $e'||c'$ where both can be reached by a common transition sequence; *common* means that both transition sequences correspond to the same sequence of events. The resulting pairs are called the *candidate pairs* of the history relation. Next, we gradually remove such pairs $\langle (q_{e'}, q), (q'_{e'}, q') \rangle$ where $(q_{e'}, q)$ has at least one predecessor via an incoming transition labeled with some event m that does not form a candidate pair with a state from $e'||c'$ where a transition labeled with m leads to $(q'_{e'}, q')$. One special case is the pair formed by the initial states of the two controllers, which we never remove. Eventually all such pairs remain where all the recent histories of the first state are also recent histories of the second. The algorithm is explained in more detail in the appendix, see Sect. B.1.

In the following, we describe how our approach is applied to our RailCab example.

5.2 Example

We assume that we are given the controller c as shown in Fig. 5 (a), which implements the MSD specification S shown in Fig. 2[4]. Furthermore, we assume that the specification was changed as described in Sect. 4.4. The controller c can be obtained as a result of manual development or a previous automatic synthesis. We assume that c was synthesized from the MSD specification; the labels in the states correspond to cuts in the MSDs of S, see Fig. 2. Transitions that are labeled with a message event where the sending object is not controlled by the are called *uncontrollable*. They are shown as dashed arrows. If the sending object of the message is controlled by the controller, we call the transition *controllable*. These are shown as solid arrows.

As a first step, we synthesize the maximal controller e' from the new environment assumptions A' of specification S' (shown on the left of Fig. 5 (a)). The environment assumptions here consist only of the assumption MSD PassingPointsOnTrack. The labels in the states of e' correspond to cuts in that MSD accordingly[5].

Given e' and c, we compute the parallel composition $e'||c$, shown on the left of Fig. 5 (c). States in $e'||c$ are labelled in the form $(q_{e'}, (q_c))$, where $q_{e'}$ is a state in e' and q_c a state in c. If in $e'||c$ there are two states q, q' such that $q = (q_{e'}, (q_c))$ and $q' = (q'_{e'}, q_c)$ (i.e., they differ only in the environment state), this means that there is an environment event in e' which is not "remembered" by the current controller c.

[4] In the subsequent figures of the controllers, the events are abbreviated as follows: `endOfTS` becomes `eTS`, `requestEnter` becomes `rE`, `enterAllowed` becomes `eA`, `approachingCrossing` becomes `aC`, `lastBrake` becomes `lB`, `lastEmergencyBrake` becomes `lEB`, `checkCrossingStatus` becomes `cCS`, `crossingStatus` becomes `cS`, `noReturn` becomes `nR`, `enterNext` becomes `eN`.

[5] To reduce the visual complexity, we do not show the self-transitions of the automata in Fig. 5. The maximal controller e' allows not only for `endOfTS` to occur in the initial state, but also for any other environment event. Complying with Def. 5, it can also receive any system message in every state. Likewise, c and c' have outgoing transitions for all environment events in the states that have outgoing uncontrollable transitions.

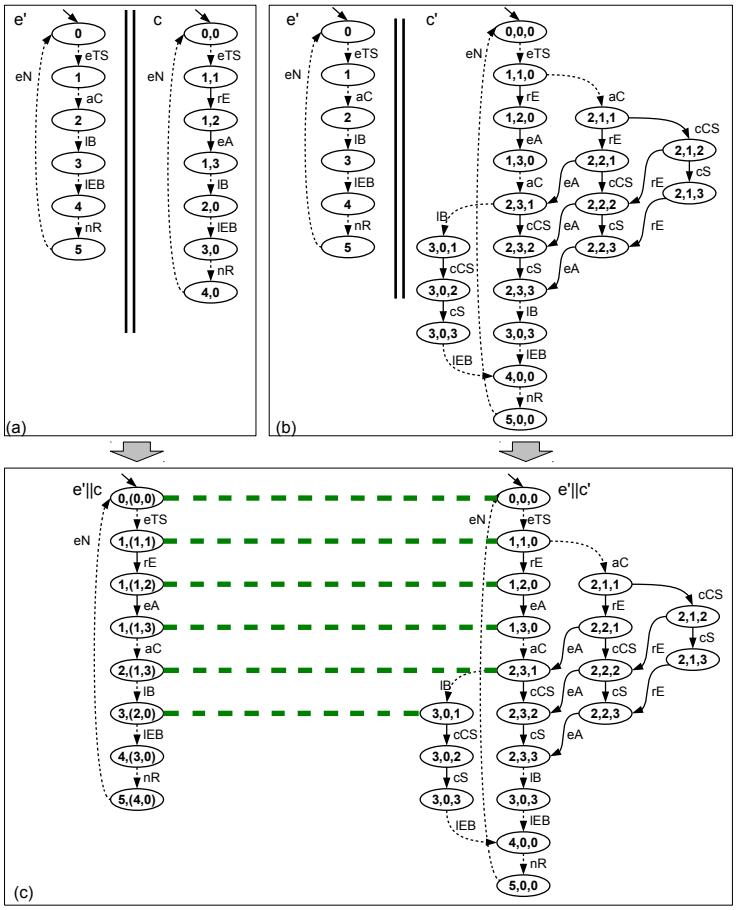


Figure 5: Synthesis Approach

In the second step, we synthesize the maximal controller c' from S'. The states of c' correspond to reachable cuts in the MSDs of S' (see that the states of c' in Fig. 5 (b) are labeled with the cuts of PassingPointsOnTrack, RequestEnterAtEndOfTrackSection, and RequestEnterAtEndOfTrackSection, in this order, as they are numbered in Fig. 3).

We then compute the parallel composition $e'||c'$ between e' and c'. The controller $e'||c'$ has the same states as c' due to the fact that c' is synthesized including the same assumption MSDs that are the basis for synthesizing e'.

In the third step we compute the history relation between the states in $e'||c$ and $e'||c'$. This relation maps states q in $e'||c$ to states q' in $e'||c'$, if for every transition sequence h leading from the initial state to q, there is a transition sequence h' in $e'||c'$ from its initial state to q' labeled with the same event sequence. The resulting mapping is shown in Fig. 5 (c) as dashed lines that connect the two controllers.

Figure 6 shows how in the last step a dynamically updating controller is formed from combining c and c' and transforming the history relation to the update transitions. The update transitions are labeled with ϵ to indicate that no observable event is associated with these transitions. We for example cre-

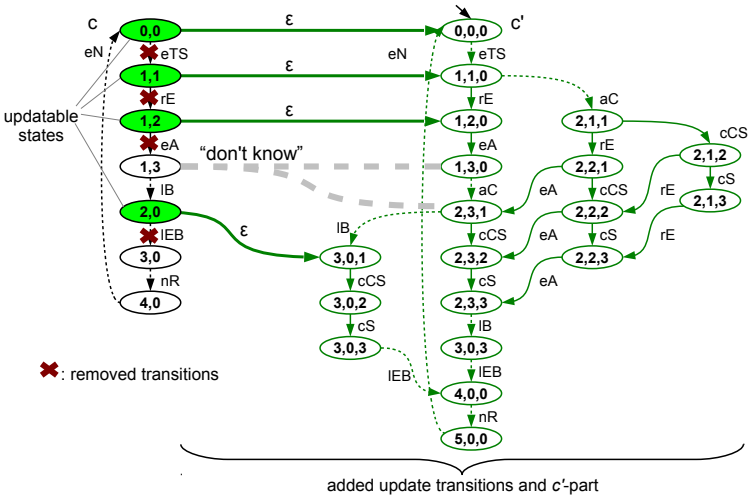


Figure 6: Dynamically Updating Controller

ate an update transition $((1,1),\epsilon,(1,1,0))$ because $((1,(1,1)),(1,1,0))$ is the unique mapping in the history relation. We do not create an update transition for state $(1,3)$, because in the history relation there are two mappings, $((1,(1,3)),(1,3,0))$ and $((2,(1,3)),(2,3,1))$ where c is in $(1,3)$, but c' is in different states $((1,3,0)$ and $(2,3,1))$. Thus, for the state $(1,3)$, there would be different continuations of the execution and therefore the state is not updatable.

Figure 6 also shows, as explained above, which transitions are deleted from and which transitions and states are added to a running instance of the current controller when installing the update.

5.3 Correctness of the Approach

In the following, we justify that our approach is *correct*, i.e. it correctly identifies updatable states and it constructs a dynamically updating controller that performs correct updates according to Def. 8. Our approach is not complete, which means that it may not successfully identify all updatable states, for reasons also discussed below.

Definitions 7 and 8 require us to consider a current controller in an environment where anything can happen as long as it satisfies the new environment assumptions. A state q in the current controller (c) is then updatable if there exists a controller that, in this new environment, first (1), can continue executing all possible recent histories of q to a run that satisfies the new specification; Second (2), this controller will continue each possible recent history of q in such a way that the continuation would also form a run that satisfies the new specification with every other possible recent history of q.

Our approach correctly identifies updatable states because it correctly shows if a state satisfies the two properties (1) and (2).

1. We synthesize a maximal controller e' from the new environment assumptions. Therefore the parallel composition $e'||c$ yields a model that allows us, as required by our definition, to derive all possible recent histories

for all states in the current controller c. We then synthesize a new controller c' for the changed specification S'. The computation of the history relation then relates states in $e'||c$ and $e'||c'$ where every possible recent history of the state in $e'||c$ is also a recent history in the states in $e'||c'$. By the way we create update transitions and thereby identify updatable states in Step 4, we can then ensure that every possible recent history of an updatable state is also a prefix of a run of the new controller in any environment satisfying the new assumptions. As the new controller implements the new specification, it will complete this prefix to a run satisfying the new specification.

2. By the way we create update transitions and thereby identify updatable states in Step 4, specifically because we only create update transitions where one state in the current controller has exactly one target state in the new controller, we guarantee that every continuation of one possible possible recent history of the updatable state will also be a valid continuation of every other possible recent history of that state (see also Lemma 1).

Performing correct updates: We consider that our procedure for "patching" the current controller to a dynamically updating controller (as briefly described at the beginning of Sect. 5.1) does not interfere with the observable behavior of the system. After installing the dynamically updating controller, the running system performs a correct update for the following reasons. If the current state (now a state in the c-part of the dynamically updating controller) is not an updatable state, the dynamically updating controller will behave as controller c until an updatable state is reached. If the current state is an updatable state or an updatable state is reached, it will immediately take the update transition to its corresponding state in the c'-part. Then, every recent history of the updatable state will be completed to a run satisfying S'.

In our approach, by synthesizing the maximal controller c' for S', we will in many cases be able to identify all updatable states. However, our approach is *not complete* due to the following problem: It may happen that the history relation may relate one state in the current controller to multiple states in the new controller. As described above, we do not create update transitions in this case, since it may not be ensured that from all these related states the system will react to future events always in the same way. However, it may be that for all these related states there exists the possibility to react to any future sequence of environment event in the same way, so that for the same sequence of environment events the controller could still produce the same continuation of the run.

Especially in a maximal controller, there may be different states from where the system has different sets of options on how to continue in satisfying the specification, but some of the options may be the same for all these states. This problem can be mapped to the problem of checking whether there exists a particular controller that instead of multiple states has only one state in the history relation for one state in the current controller (see Lemma 1). This problem is not trivial, and we plan to address it in future work. In most cases, however, the above approach will identify all updatable states.

6 Incremental Synthesis of Dynamically Updating Controllers

The systematic approach described in the previous section requires the synthesis of a maximal controller c' from the new specification, and the synthesis of the maximal controller e' from the environment assumptions. However, the synthesis of the maximal controller is intrinsically expensive.

In this section, we propose a more efficient approach , based on a novel algorithm with two peculiarities. First, it is an *on-the-fly* algorithm (inspired by [10]), which means that it can synthesize a controller from an MSD specification without exploring all reachable cut configurations. Second, it is an *incremental* algorithm, which means that a new controller for a changed specification can be synthesized based on the current implementation. This makes the synthesis more efficient, as will be explained in more detail shortly. Most important for our purposes, the resulting controller will be very similar to the current controller, which in many cases allows us to compute the maximal set of updatable states. At the same time, compared to the result of the systematic approach, the resulting new controller will usually be much smaller and it will be deterministic, which is typically desired for the implementation.

The introduction of the new algorithm considerably simplifies the overall approach for synthesizing the dynamically updating controller, which now can be divided into three main steps:

1. Given the current controller c implementing the specification S, and the new specification S', we apply the on-the-fly incremental algorithm to synthesize the new controller c'.

2. The incremental algorithm builds a correspondence between states in c and c'. A subset of these corresponding states is used for computing the history relation. Different from the systematic approach the synthesis of the environment controller e' and the respective computation of the parallel composition $e'||c$ and $e'||c'$ is not required anymore.

3. As for the systematic approach, we identify updatable states and derive update transitions from the history relation.

In a nutshell, the on-the-fly synthesis algorithm works as follows. From the start state, the algorithm checks if the system can always guarantee to reach a *goal* state. A goal state is a state where no safety violation occurred in the requirements and no active events remain in active requirement MSDs. A state is also a goal state if a safety violation occurred in an assumption MSD, or if active events remain in an active assumption MSD, i.e., if the environment violated the environment assumptions or there are still events that we assume will happen eventually. From these goal states the algorithm will then check if the system can again guarantee reaching a goal state, etc., so that at the end it is ensured that the system can reach goal states infinitely often. If this is the case, we can extract a controller that implements the specification.

Whether the system can guarantee to reach a goal state from some other state is checked by a depth-first exploration and backward labeling of states. States are labeled as *winning* if a goal successor or another winning successor can be reached by at least one controllable transition or by all uncontrollable

transitions. For the synthesis, we assume that the system can only send messages according to active system messages in requirement MSDs. This implies that the system must wait for the next environment event if there are no active messages. However, if there are active events in requirement MSDs, the system can also wait for the next environment event.

This algorithm can be turned into an incremental algorithm intuitively as follows. If the specification is consistent, and thus there exists a controller that implements the specification, the synthesis may, by chance find such a controller without ever exploring any "dead ends", where it finally it turns out reaching another goal state cannot be guaranteed. In other words, if the specification is consistent, this implies that, when the system is active, there is always one "smart" choice by which the system can successfully satisfy the specification.

We also call this smart choice the *winning action*. This winning action is typically sending one particular system message. In some cases the winning action can also be to not send any system message, but to wait for the next environment event. The idea of the incremental algorithm is to help the algorithm in making smart choices based on the winning actions in a given current controller.

To do this, while exploring the state space, the incremental algorithm first keeps track of which states that it currently explores *correspond* to which states in a given current controller. As we will see shortly, this correspondence relationship can be many-to-many. Then, if there are active messages in a state, the algorithm will continue exploring by first trying the winning actions of the corresponding states in the current controller. If it turns out that the old winning actions do not guarantee the system to always eventually reach goal states, the synthesis algorithm will try another choice. From that point forward, however, no correspondence to states in the current controller can be maintained anymore, so the algorithm works as before.

Intuitively, states of the current controller correspond to states in the newly explored state graph under the following conditions. First, when they can be reached by a common sequence of events or, second, when, somewhere in between, the current controller did something that was not specified in the new specification, and is therefore not of its concern. Third, states correspond also if somewhere in between the new specification requires the system to do something which was not of concern to the current controller. More precisely, the states correspond according to the following conditions: First, the start state of the newly explored state space corresponds to the initial state of the current controller. Then, if the state q in the current controller corresponds to q' in the newly explored state graph, also the following states correspond. (Σ is the alphabet of message events in the old specification; Σ' is that of the new specification.)

1. If q and q' both have outgoing transitions labeled with the same event, the target states of these transitions are also corresponding. (This also applies if one of the target states of these transitions is q or q', i.e., if transitions are self-transitions.)

2. If an outgoing transition of q is labeled with an event that is not element of Σ', the target state of the transition leaving q also corresponds to q'.

3. If an outgoing transition of q' is labeled with

(a) an environment event that is not element of Σ, or

(b) a system event that is not element of Σ,

the target state of the transition leaving q' also corresponds to q.

The details of this synthesis procedure are described in the Appendix A.

The second step after the incremental synthesis of the controller c' for the new specification is to compute the history relation between the current controller c and c'. As described in Sect. 5.1, the history relation is computed on the basis of candidate pairs. Our systematic approach required us to compute these candidate pairs explicitly on the basis of the parallel compositions $e'||c$ and $e'||c'$. Here instead, the candidate pairs do not have to be computed explicitly: they are already contained in the set of the corresponding states resulting from the incremental synthesis.

Candidate pairs are formed on the basis of the same conditions as corresponding states (see above), except condition 3b. This condition would allow for forming a candidate pair where in the new controller a system event occurs for which we know that it did not occur in the current controller. Therefore, there would be pairs without a common recent history[6]. Because the candidate pairs can be formed using these conditions, building the parallel composition of $e'||c$ is no longer required.

As an example, Fig. 7 shows the resulting controller c' from synthesizing our changed example MSD specification S' from Fig. 3. Also it shows the base controller c, the corresponding states, and the subset of corresponding states that form candidate pairs for computing the history relation. The resulting controller is a reduced version of the maximal controller in Fig. 5; we keep the topology of the states to highlight this. See that transitions with the same events are explored from corresponding states. Only in state (1,3,0) we wait for the next environment event, because in the corresponding base controller state there are no outgoing transitions with system events, i.e., the winning action is to wait for the next environment event. The corresponding states after the new controller sends `cCS` (`checkCrossingStatus`) are not candidate pairs anymore.

Our synthesis algorithm, in states where the system decides to wait for the next environment event, will always explore all environment events. If this does not lead to a state change, this results into self-transitions, which are not shown here. Environment events that lead to violations in the environment assumptions result in transitions to a sink state that represents that an environment assumption occurred. In Fig. 7, this state is shown on the right, labeled "av".

Finally, based on the candidate pairs, we can now compute the history relation and create the dynamically updating controller as before (see Sect. 5.1 and 5.2).

[6] Note that the converse is not a problem: If in the current controller a system event occurs that is not subject to the new specification, there could exist a controller implementing the new specification that performs these action at any time. By the conditions for forming the candidate pairs, we thus do indeed relate states where q in the current controller may have a recent history that is not a recent history of the counterpart state in the synthesized new controller q'. However, we know that there *exists* an alternative new controller with a state that has the desired recent histories, and then continues as the synthesized new controller in state q', which is sufficient for what Def. 8 requires.

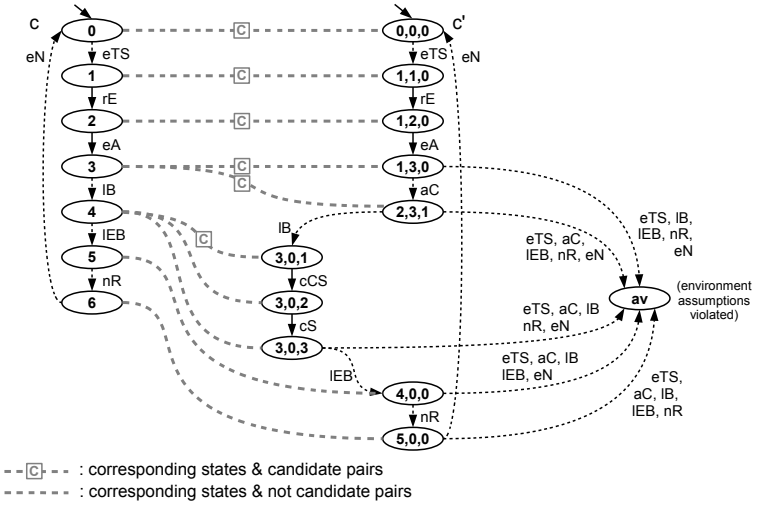


Figure 7: Incrementally synthesized controller with corresponding states and candidate pairs

7 Tool and Validation

ScenarioTools is a set of Eclipse plug-ins that support the engineer in modeling and simulating MSD specifications. Furthermore, controllers can be synthesized from MSD specifications. For this paper, we extended ScenarioTools with the capability to incrementally synthesize dynamically updating controllers as described in Section 6.

In ScenarioTools, an MSD specification is modeled in UML which is extended with a profile that adds the temperature and the execution kind to messages. For modeling, we use the Papyrus UML editor[7], which we extended to color hot messages red, cold messages blue, and display dashed arrows for monitored messages. Figure 8 shows a screenshot of the extended Papyrus sequence diagram editor.

Both synthesis and simulation in ScenarioTools use the same run-time logic. This common semantic basis ensures consistent results for both kinds of analysis. Also the synthesis and simulation both support the same rich MSD features.

Figure 9 shows an overview of the process for performing synthesis and simulation with ScenarioTools. It visualizes the most important models involved (marked with numbers 1 to 4) and their relation. First, the engineer models the MSD specification (1), then ScenarioTools maps the specification's class diagrams to a corresponding Ecore class model (2). From this model, and the collaboration that describes the object structure in the UML model, a system of objects (EObjects) is created (3). Based on these objects, and by help of the mapping from the objects to the roles in the UML model, the MSDs can be interpreted for simulation and synthesis (4).

In ScenarioTools, it is possible to systematically explore the state space described by an MSD specification. This state exploration mechanism is used

[7] http://www.papyrusuml.org

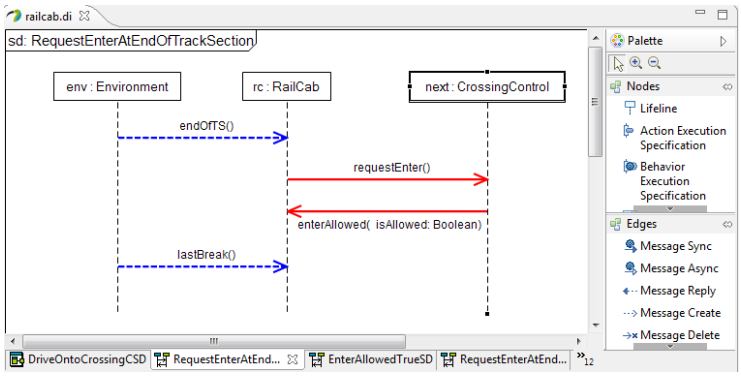


Figure 8: An MSD in the Papyrus UML-editor, extended by a ScenarioTools plug-in

by our on-the-fly synthesis algorithm described in Sect. 6. The algorithm either determines that no valid implementation exists for the given specification, or it generates a possible controller fulfilling the specification. The controllers can be serialized using the standard EMF facilities. Furthermore, we can visualize these controller models using Graphviz.

Figure 10 shows the controller generated for the example specification S, shown in Figure 2. The controller's starting state is labeled with "0". The dashed arrows indicate uncontrollable transitions. The solid arrows indicate controllable transitions. The transition labels define the message that is exchanged and the sending and receiving objects. Note that state "7" is special as it is reached only due to violations of the environment assumptions (see also Fig. 7).

Figure 11 shows the dynamically updating controller created by our tool. It was generated for the update from the original MSD specification S, as shown in Figure 2, to the modified MSD specification S', as shown in Figure 3. It was incrementally synthesized based on the controller for specification S that is shown in Figure 10. To reduce the visual complexity of the figure, we removed any states corresponding to violated environment assumptions.

8 Related Work

Dynamic software updates have been studied in the past. The problem has been addressed in the area of programming languages [8, 16, 17] and from the perspective of distributed, reactive systems [22, 7, 3, 31, 26].

The early work on dynamic software updates in the area of programming languages required that procedures affected by the changes were currently idle [8, 16]. Later, Gupta et al. [17] defined that an update of a program is valid if the current run-time state of the old program is also a reachable state of the new program. This problem is called the *state mapping problem*. The intuitive motivation for the state mappings is very similar to our motivation for updatable states: "an online change is valid, if after [...] a change the process starts behaving as if it had been executing the newer version of the program since the beginning from its initial state." [17, p. 122]. Our motivation is similar, but we

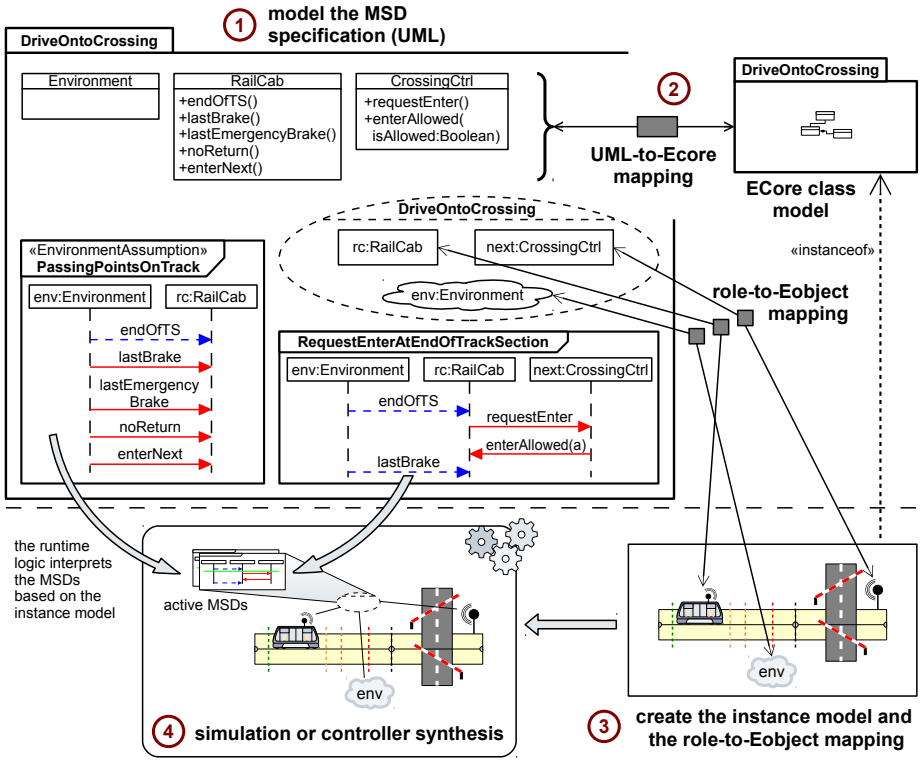


Figure 9: Overview of Synthesis and Simulation with ScenarioTools

consider the states of different finite state machines and system specifications and more generally argue over the sequences of events.

Dynamic updates in component-based systems were studied before [22, 31, 26], but these approaches do not consider the validity of updates with respect to specification changes. Chaki et al. define that a component can be updated if it still provides the services of the old [7], but this implies that the specification cannot become more restrictive.

In the area of dynamically adaptable systems, a number of techniques for modeling and verifying adaptive software have been elaborated [14, 33, 2, 6, 12] The languages they propose allow for specifying software that can reconfigure between a fixed set of configurations at pre-defined update points. Giese et al. propose a formalism based on state charts and regard mainly the reconfiguration of continuous controllers [14]. Zhang et al. propose a formalism for modeling adaptive software that requires the manual definition of update points [33]. They provide a specification language and verification support for temporal properties that are invariant during the adaptation or adaptation-specific. Adler et al. propose a framework for developing dynamically adaptive embedded systems [2], and Fisher et al. propose a formal language for modeling adaptive software [12]. Bouveret et al. describe a categorical framework to ensure correct software evolutions [6]. They present a formalism-independent framework that captures distributed architectures and patches and allows for identifying obligations for proof that then need to be carried out. These approaches, how-

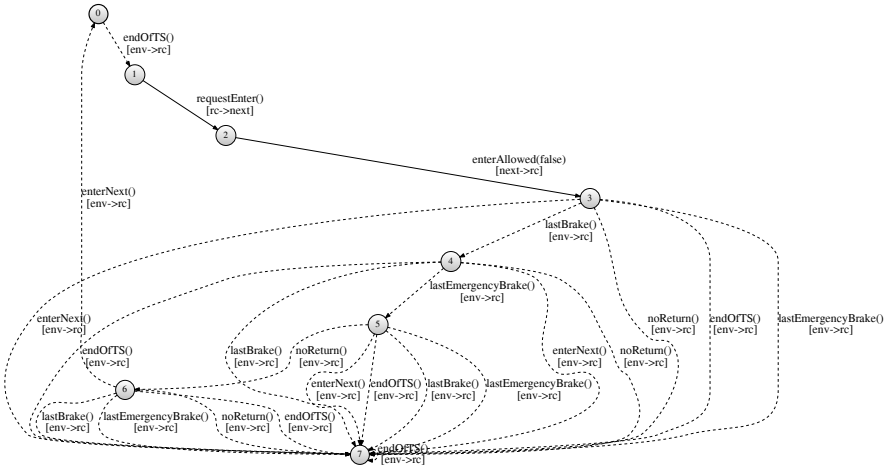


Figure 10: Global Controller

ever, do not consider that certain configurations comply to certain requirements and that reconfigurations must satisfy certain conditions with respect to these requirements.

Hayden et al. [21] present an approach for testing dynamically updating software. It assumes that test suites for two program versions are given and performs tests of the software before and after an update. Also here the update points are specified manually and the allowed update behavior is implied by the test suites—criteria for allowable update points or automatically finding these points is not considered, nor is any relation defined between update tests and specifications changes.

Anderson and Rathke [4] study updates of protocols in multi-threaded systems. They present an approach for analyzing how it can be guaranteed that all threads update in such a way that the whole system switches to a new protocol and not any threads behave according to the old protocol. Here again, the new protocol and update points are implemented manually and no criteria are considered for when a protocol update is correct with respect to the changed requirements. However, the issues discussed by Anderson and Rathke are relevant also for synthesizing distributed dynamically updating controllers, which is an outlook of this paper.

9 Conclusion and Outlook

This paper takes a specification-oriented perspective on dynamic software updates. We considered the question of when a dynamic update of a controller is correct with respect to changes in its specification. This is crucial because changes in practice are mostly considered on the specification level first.

In our previous work, we introduced a formal criterion for correct dynamic updates and a conceptual approach for automatically synthesizing dynamically updating controllers from changes in scenario-based specifications [13]. In this paper, we extended our previous work by introducing a novel and more efficient

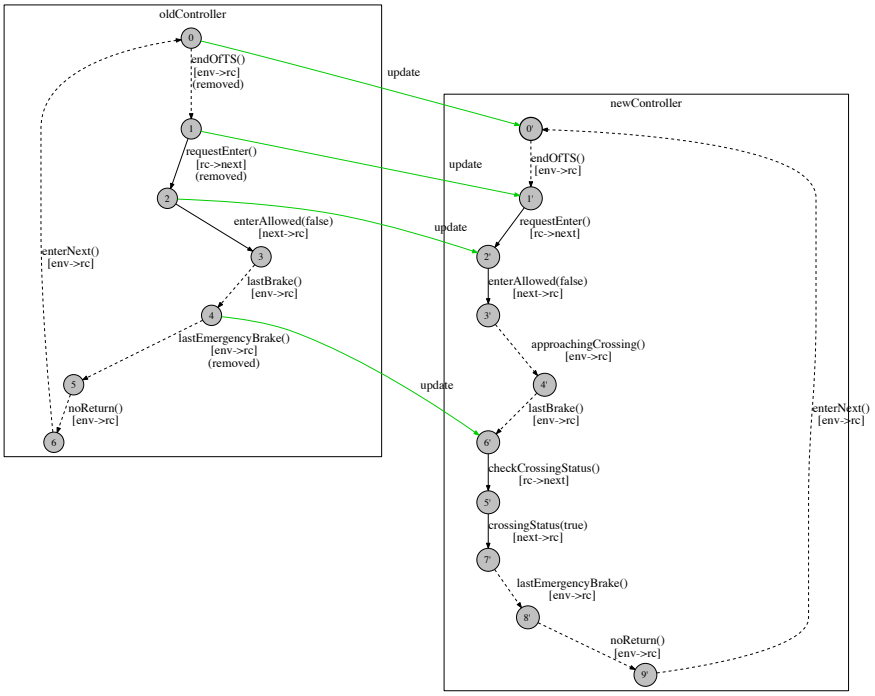


Figure 11: Dynamically Updating Controller Based on Controller in Figure 10

algorithm for synthesizing dynamically updating controllers.

The key idea of the algorithm was to synthesize a dynamically updating controller from the specification change *on-the-fly* and *incrementally*, based on the current controller. The proposed algorithm favors the practical applicability of our techniques by reducing the inherent complexity of the synthesis problem in the typical cases of evolutionary specification changes. Moreover, we provided an implementation of the approach as part of ScenarioTools, a novel Eclipse-based synthesis and simulation tool suite for MSD specifications. Although we considered the specification to be formalized by MSDs, the approach can also be adapted to other specification formalisms like linear temporal logic or automata on infinite words.

Our approach is also a first step towards building self-adaptive systems where the system autonomously derives new requirements at run-time, e.g., from high-level goals, or learns about changing environment properties.

In the future, we plan to extend the approach to support the synthesis of a distributed dynamically updating controller for each system object. This is possible in principle [18], but rigorous techniques for doing so are still subject to current research.

A APPENDIX: Controller Synthesis from MSD Specifications

In the following we describe the algorithm for synthesizing controllers from MSD specifications and its incremental extension.

A.1 On-The-Fly Synthesis Algorithm

The problem of synthesizing a controller can be viewed as the problem of finding a winning strategy in a two-player game, played by the system against the environment. In this game, the environment can choose to send environment messages and the system can choose to send system messages. These messages activate, progress, terminate, or violate the MSDs in a specification as described in Sect. 4. In a state with active (enabled executed) messages, the system has priority over the environment in choosing to send one of the active messages. In our case, we do not consider that the system can send non-active messages, i.e., if there are no active message, the system waits for the next environment event to occur. The system can also decide to wait for the next environment event even if there are active messages. If the system waits, the environment can send an arbitrary environment message; it cannot be inactive indefinitely. From this follows that the game always progresses. A state graph where states represent sets of active MSDs in particular cut configurations and transitions represent environment or system moves is called a *game graph* in the following.

The system wins the game if the sequence of sent messages results in a run that satisfies the MSD specification. We translate this to the following winning condition. The system wins the game if it can guarantee to *infinitely often* reach a state with the following property, also called *goal* property: (1) there must not have occurred a safety violation in any requirement MSD or there are active messages in at least one assumption MSD *and* (2) there must be no active messages *or* (3) there was a safety violation in an assumption MSD. In other words, the system wins if it can always eventually reach a state where it did not violate any requirements and it has no unfulfilled obligations and waits for the environment; furthermore, it wins if there was a safety violation of the environment assumptions or, if a safety violation occurred in the requirements, subsequently there will always remain open environment obligations. States fulfilling this property are called *goal states* in the following.

Games with a winning condition requiring to infinitely often visit goal states are called *Büchi games* [28]. David et al. propose an efficient *on-the-fly* algorithm for solving Büchi games in Uppaal Tiga [10, Sect. 6.4.5] that has the advantage that in many cases this algorithm will find a winning strategy without exploring the complete state space. The algorithm is based on an algorithm for minimal fixpoints by Liu and Smolka [25]. We adopt the algorithm by David et al. in ScenarioTools for incrementally and non-incrementally synthesizing controllers from MSD specifications. In the following, we first explain the non-incremental algorithm and then explain a slight extension by which this algorithm can be made incremental.

The algorithm for solving the Büchi game is based on an algorithm for solving reachability games. In a reachability game we ask whether the system can guarantee to eventually reach a goal state although the environment tries

everything to keep the system from doing so. Roughly, the Büchi game is then solved as follows: First, it is checked whether the system can guarantee to reach a goal state from the initial state of the game, which corresponds to a state where no event has yet occurred and there are no active MSDs. If this is not possible, we know the system will lose. If this is possible, there may remain goal states from which we do not know whether it is possible to again reach another goal state, so we call the reachability procedure again for these states. Eventually, we may find that for every goal state we can reach another goal state, so the system can satisfy the winning condition. But it may also be that for some goal state we find that we cannot guarantee reaching another goal state. At this point, we mark this state as *loseBüchi* and check for all its predecessors if reaching a goal state can still be guaranteed. If in the end we must mark also the initial state as *loseBüchi*, the system loses the Büchi game.

Algorithm 1 shows the procedure for solving the reachability game, OTFR ("On-The-Fly Reachability"), and Alg. 2 shows the procedure for solving the Büchi game, OTFB ("On-The-Fly Büchi"). The algorithms are shown in a Java-like pseudo-code, with the Java-like types Set, Map, Stack, and the typical operations on them. Where appropriate, we also use the notation of the usual mathematical operations on sets. Furthermore, it is assumed that we can navigate from transitions to source and target states by the operations getSourceState() and getTargetState().

Let us first consider in more detail the OTFR procedure in Alg. 1. The procedure takes a state of a game graph as input. This state can be the start state of the game graph, but within the OTFB procedure, we may also call OTFR with another state as start state. So, in the context of the OTFR algorithm, the term start state always refers to the state that we give as input. The procedure returns *true* if the system can guarantee to reach a goal state from the start state, and $false$ otherwise. The algorithm by David et al. [10] would return true immediately if the start state is already a goal state, but we require that goal states must be reachable by taking at least one transition. The algorithm also works on a range of variables that we declare as global variables, because, as explained shortly, they are also accessed by the OTFB procedure and across multiple invocations of OTFR.

- *Passed* is a state set to mark the states that were already visited.
- *Waiting* is a stack for the depth-first exploration of states.
- *loseBüchi* is a set of states to mark the states for which it is known that reaching a goal state infinitely often cannot be guaranteed.
- *Win* is a set of winning states from where the system can guarantee to win a reachability game, i.e., from where it can guarantee to eventually reach a goal state.
- *Goal* is a set of states to mark explored states that fulfill the goal condition.
- *Depend* is a map that maps states to their incoming transitions from visited predecessors.

After calling the procedure, it will mark the start state as passed and push all outgoing transitions of the start state to the *Waiting* stack to schedule

them for further exploration (this is done by the procedure **pushOutTransitionsOnStack** that is not explained here in more detail). Then the algorithm enters a loop where it does the following two things: It will *explore forward* the state space in a depth-first way and, if it visits a goal state, it will try to *backward re-evaluate* the predecessor states to determine whether for the predecessor states reaching a goal state can be guaranteed. On a successful backward re-evaluation of a predecessor state, this state is marked as winning and the backward re-evaluation continues, possibly until the start state is marked winning.

The forward exploration takes place in the if-branch of the while-loop in lines 15 to 22. The algorithm marks newly explored target states as passed and adds the explored transition to the *Depend* map entry of the target state. Then the algorithm schedules the currently explored transition for a backward re-evaluation by pushing it again onto the $Waiting$ stack under the following conditions: First, the state must be a goal state or a winning state. The procedure **isGoal** determines that a newly explored state fulfills the goal condition; it will also add this state to the *Goal* set. If the Win set contains a state, it means that it was already marked winning during a previous backward re-evaluation and we can guarantee that the system can reach another goal state from this state. Additionally, the state must not be marked as *loseBüchi*, but we will get to this later. If the transition is not scheduled for backward re-evaluation, the algorithm continues the forward exploration of the state space by adding all outgoing transitions of the target state to the $Waiting$ stack.

The backward re-evaluation takes place in the else-branch in lines 23 to 30. This branch is entered whenever the transition popped from the stack leads to a state that was already visited previously. This can happen because the algorithm has previously scheduled the transition for backward-reevaluation as explained above or because it explores a back-transition. In any case, the explored transition is added to the *Depend* map entry for the target state. Then it is checked whether from the source state the system can guarantee reaching a goal or winning successor state that is not marked as *loseBüchi*. This is checked by the procedure **guarGoalOrWinAndNotLoseBuechiSucc**, which we don't explain in more detail. If the system can guarantee reaching such a state, then the source state is marked as winning and all incoming edges from previously visited predecessors are scheduled for backward re-evaluation. Furthermore, once a state is marked as winning, we can stop forward exploration from this state; the procedure **removeOutTransitionsFromStack** therefore removes all outgoing transitions of that source state from the stack.

The while-loop terminates once the source state is marked winning or the $Waiting$ stack is empty. If the start state is winning, the procedure returns *true*; otherwise it returns $false$. Note that only such states are considerd *winning* by which a goal and not *loseBüchi* state can be reached by taking at least one transition. So, a goal state is only marked winning if it can again be guaranteed to reach a goal (and not *loseBüchi*) state from it. The procedure will terminate because a transition will only be added to the $Waiting$ stack at most twice. From this also follows that the complexity of the reachability algorithm is linear with respect to the number of states and transitions in the game graph. The correctness of the reachability algorithm follows from the fact that after every iteration of the while loop, every state added to the Win set is actually winning in the above sense, so if the start state is added to the Win set, upon which the

while-loop will terminate, the start state is actually winning.

Algorithm 1 On-the-fly Algorithm for Büchi Games (OTFB) (Part 1, OTFR procedure)

```
1:  variables:
2:  Set<State> Passed;   // a set of visited states
3:  Stack<Transition> Waiting;   // a stack of transitions waiting to be explored
4:  Set<State> loseBüchi;   // from where reaching inf. goal states is not guaranteed
5:  Set<State> Win;   // a set of winning states
6:  Set<State> Goal;   // Goal states added in isGoal() or guarGoalOrWinNext()
7:  Map<State, Set<Transition>> Depend;   // state to transitions from predecessors
8:  procedure OTFR(startState)
9:    Passed.add(startState);
10:   pushOutTransitionsOnStack(startState, Waiting);
11:   while !Waiting.isEmpty() ∧ !Win.contains(startState) do
12:     Transition t = Waiting.pop();
13:     State sourceState = t.getSourceState();
14:     State targetState = t.getTargetState();
15:     if !Passed.contains(targetState) then   // forward exploration
16:       Passed.add(targetState);
17:       Depend.get(targetState).add(t);
18:       if isGoal(targetState) ∨ Win.contains(targetState)) ∧
                    !loseBüchi.contains(targetState) then
19:         Waiting.push(t);
20:       else
21:         pushOutTransitionsOnStack(targetState, Waiting);
22:       end if
23:     else   // backward re-evaluation
24:       Depend.get(targetState).add(t);
25:       if guarGoalOrWinAndNotLoseBuechiSucc(sourceState) then
26:         Win.add(sourceState);
27:         Waiting.pushAll(Depend.get(sourceState));   // for bwd re-eval
28:         removeOutTransitionsFromStack(sourceState, waiting);
29:       end if
30:     end if
31:   end while
32:   return Win.contains(startState);
33: end procedure
```

The OTFR procedure may be invoked several times from the OTFB method shown in Alg. 2. The OTFB procedure takes as input the start state of the game graph and returns *true* if the system can guarantee to always eventually reach a goal state or $false$ otherwise. It works on the global variables mentioned above as well as the local stack variable *Reevaluate* that contains states for which the algorithm yet has to reevaluate whether reaching a goal and not *loseBüchi* state can be guaranteed. The algorithm works as follows.

First, the algorithm checks whether reaching a goal state from the start

state is possible (line 36). If not, the algorithm returns *false* immediately. Otherwise, the result is that the game graph is partially explored, a subset of states is marked winning and a subset of states is identified as goal states. As long as there exist goal states that are not marked winning or *loseBüchi*, the algorithm iterates in the while loop in lines 39 to 54. In each iteration, one of these states is added to the *Reevaluate* stack, which will always be empty at the beginning of each iteration in this while-loop. Then, as long as the *Reevaluate* stack is not empty, the while loop in lines 42 to 53 iterates.

In each iteration of this inner loop, OTFR is called for one state that is popped from the *Reevaluate* stack. If OTFR returns *false*, then we know that for this state the system cannot guarantee to reach a goal or winning and not *loseBüchi* state. Therefore, we then also mark this state as *loseBüchi* and push all its visited predecessors to the *Reevaluate* stack. If OTFR *returns* false for the start state of the game graph, we return *false* immediately. If OTFR returns *true*, it means that from this state the system can guarantee to reach a goal or winning and not *loseBüchi* state. In this case, the predecessors need not be pushed to the *Reevaluate* stack.

The procedure will terminate for the following reasons. In the inner loop, new states are only added to the *Reevaluate* stack if they are not marked *loseBüchi* and also at least one state is marked *loseBüchi*. Since the game graph is finite and states never lose the *loseBüchi* status, the inner while loop terminates. The outer while loop terminates because after each iteration at least one goal state is either marked winning or *loseBüchi*. The result is correct, because the procedure returns *false* if the start state is marked *loseBüchi*; otherwise, if the procedure returns *true*, we know that the start state is winning and all reachable goal states that are not *loseBüchi* are also winning.

The repeated calls to the OTFR procedure do not have such a big impact on the run-time as it seems: Once the complete game graph was explored, calls to the OTFR procedure boil down to just calling the **guarGoalOrWinAndNotLoseBuechiSucc** procedure to re-evaluate the winning status of a state. However, states can in the worst case be pushed and popped from the *Reevaluate* stack a quadratic number of times. But this is only the case if all states are tightly interconnected, which is rarely the case.

If the OTFB procedure returns *true*, we can extract a controller from the states that are marked winning and not *loseBüchi* and the transitions between. In the resulting state graph, however, there may still exist cycles that do not contain any goal state, and which need to be removed. To do this, we first remove such transitions that close a loop in the controller. These transitions we detect using a depth-first-search algorithm. As this can lead to states without outgoing transitions, we need to remove those as well, including all their incoming transitions. We do this repeatedly until reaching a state in the cycle which has another outgoing transition.

A.2 Incremental version of the algorithm

The extension of the algorithm to make it incremental works as follows. We assume that as input we are given a *base controller* c and a specification S' for which we want to synthesize a controller. The alphabet Σ of c can be different from the alphabet Σ' of S', but the intersection must not be empty.

Initially, the initial state of c *corresponds* to the initial state of the game

Algorithm 2 On-the-fly Algorithm for Büchi Games (OTFB) (Part 2, OTFB procedure)

```
34: procedure OTFB(startState)
35:   Stack<State> Reevaluate;
36:   if !OTFR(startState) then
37:     return false;
38:   end if
39:   while Goal \ (Win ∪ LoseBuechi) ≠ ∅ do
40:     State q ∈ Goal \ (Win ∪ LoseBuechi);
41:     Reevaluate.add(q);
42:     while !Reevaluate.isEmpty() do
43:       State q2 = Reevaluate.pop();
44:       Win.remove(q2);    // so that OTFR does not terminate immediately.
45:       if !OTFR(q2) then    // else q2 will immediately be re-added to Win.
46:         loseBüchi.add(q2);
47:         q2 == startState : return false;
48:         for all Transition t : Depend.get(q2) do
49:           State src = t.getSourceState();
50:           !loseBüchi.contains(src) ∧ src != q2 : Reevaluate.push(src);
51:         end for
52:       end if
53:     end while
54:   end while
55:   return true;
56: end procedure
```

graph induced by S'. As we explore further states in the game graph, the successor states again correspond under certain conditions and if states correspond, we modify the order in which their outgoing transitions will be explored:

Corresponding successor states: If two states q and q' are corresponding, also the following states are (if they are explored at all by the on-the-fly algorithm).

1. If q and q' both have outgoing transitions labeled with the same event, the target states of these transitions are also corresponding. (This also applies if one of the target states of these transitions is q or q', i.e., if transitions are self-transitions.)

2. If an outgoing transition of q is labeled with an event that is not element of Σ', the target state of the transition leaving q also corresponds to q'.

3. If an outgoing transition of q' is labeled with

   (a) an environment event that is not element of Σ, or

   (b) a system event that is not element of Σ,

   the target state of the transition leaving q' also corresponds to q.

Order of exploring outgoing transitions: If two states q and q' are corresponding, schedule the outgoing transitions of q' to be explored in the following order.

1. Transitions with a system event for which q has a corresponding outgoing transition labeled with the same event.

2. Transitions with an environment event for which q has a corresponding outgoing transition labeled with the same event.

3. Transitions labeled with an environment or system event for which q has no corresponding outgoing transition labeled with the same event.

Technically, the order in which transitions will be explored by the synthesis algorithm presented above can be controlled in the procedure **pushOutTransitionsOnStack** (see Alg. 1, line 10 and 21).

B APPENDIX: Computing the History Relation

B.1 Algorithm for the Computation the History Relation (HRA)

Algorithm 3 shows the pseudo-code for computing the history relation between two controllers. The algorithm consists of one procedure, called HRA which takes as input two controllers c, c', and two states defining the starting point of the algorithm, and returns a set $History$ of pairs defining a relation between states in c and in c'. If we invoke the procedure for the specific controllers $e'||c$, and $e'||c'$, starting from their respective initial states, the resulting $History$ set contains all the state pairs belonging to the history relation. In our example, the result of the algorithm is shown in Fig 5 (c) as green-dashed lines.

The algorithm is shown in a Java-like style as above, additionally we introduce a short-hand notation $q_1 \xrightarrow{m} q_2$ for transitions where q_1 and q_2 are the source and target states, and m is the message event labeling the transition.

Intuitively the algorithm works as follows. We first populate $Candidate$ (line 5) with all possible state pairs that can potentially be part of $History$. Then we iteratively remove all the unnecessary pairs until $Candidate$ contains only the pairs in the history relation.

The $Candidate$ set contains pairs of states belonging respectively to controllers c and c'. A pair $<q, q'>$ is introduced into $Candidate$, if its elements q and q' are target states of a transition labeled by the same event. The computation of the candidate set is performed by the procedure ComputeCandidate described in Algorithm 4. The procedure performs a forward exploration starting from the pair corresponding to the initial states of the controllers. At each iteration the visited pair $<s, s'>$ is the source of a further exploration. The exploration is driven by the outgoing transitions of state s. Given a transition t from s to a state q of controller c, the algorithm looks for a state q' of c' reached by a transition starting from s' that is labeled with the same event associated to t. If such a state exists the pair $<q, q'>$ is added to the $Candidate$ set and explored in the next iteration. The procedure terminates when there are no other pairs to be explored.

The $Candidate$ returned by the procedure ComputeCandidate contains all the pairs that have at least one incoming transition labeled with the same event. However, given a pair $<q, q'>$ in the $Candidate$ set, the state q could also have incoming transitions that are labeled with an event that is not present

Algorithm 3 Algorithm for the Computation the History Relation (HRA)

```
1: variables:
2: Set<Pair<State, State>> History;
3: Set<Pair<State, State>> Candidate;
4: procedure HRA(c, c', q_start, q'_start)
5:   Candidate = ComputeCandidate(c, c', q_start, q'_start);
6:   bool newCandRemoved = true;
7:   while Candidate != {< q_start, q'_start >} ∧ newCandRemoved do
8:     for all Pair<State, State> <q, q'> : Candidate \ {< q_start, q'_start >} do
9:       newCandRemoved = false;
10:      for all transitions s --m--> q in c, s being some state of c do
11:        if (∄s' of c' s.t. s' --m--> q')
                    ∨ (∃s' of c' s.t. s' --m--> q' ∧ <s, s'> ∉ Candidate) then
12:          Candidate.remove(<q, q'>);
13:          newCandRemoved = true;
14:        end if
15:      end for
16:    end for
17:  end while
18:  History = Candidate;
19:  return History;
20: end procedure
```

Algorithm 4 Algorithm for the Computation of the Candidate Set

```
1: variables:
2: Set<Pair<State, State>> Candidate;
3: Set<Pair<State, State>> ToVisit;
4: procedure ComputeCandidate(c, c', q_start, q'_start)
5:   ToVisit = < q_start, q'_start >;
6:   Candidate = {};
7:   while !ToVisit.Empty() do
8:     Pair<State, State> <s, s'> = ToVisit.getAndRemove();
9:     for all transitions s --m--> q in c, s being some state of c do
10:      if ∃q' of c' s.t. s' --m--> q' then
11:        if <q, q'> ∉ Candidate and q ≠ c.getStartState() then
12:          ToVisit.add(<q, q'>);
13:        end if
14:        Candidate.add(<q, q'>);
15:      end if
16:    end for
17:  end while
18:  return Candidate;
19: end procedure
```

in the set of incoming transitions of q'. This means that the state q has at least one recent history that is not a recent history of q'. The pair $<q, q'>$ must not be part of the resulting *History* set and needs to be removed from *Candidate* (see line 11 of Algorithm 3). The while loop terminates when no other state pairs are removed from *Candidate*. All the remaining pairs are then inserted to *History*.

Algorithm 3 can be easily adapted to compute the history relation in the case of the incremental synthesis of controller c'. As described in Sect. 6, the set of the corresponding states resulting from the incremental synthesis already contains all the candidate pairs for computing the history relation . To compute the history relation for the incremental synthesis, the HRA procedure must take as input also the set of candidate pairs. Moreover, since the set of candidate pairs has been already computed by the incremental synthesis algorithm, the procedure ComputeCandidate is not invoked. As for the original algorithm, the *History* set is obtained by an iterative removal of candidate pairs.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[2] R. Adler, I. Schaefer, T. Schuele, and E. Vecchié. From model-based design to formal verification of adaptive embedded systems. In *Proc. 9th Intl. Conf. on Formal methods and software engineering*, ICFEM'07, pages 76–95, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] S. Ajmani. *Automatic software upgrades for distributed systems.* PhD thesis, MIT, Cambridge, MA, USA, 2004.

[4] A. Anderson and J. Rathke. Migrating protocols in multi-threaded message-passing systems. In *Proc. 2nd Intl. Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 8:1–8:5, New York, NY, USA, 2009. ACM.

[5] Y. Bontemps and P.-Y. Schobbens. Synthesis of Open Reactive Systems from Scenario-Based Specifications. In *Proc. 3rd Int. Conf. on Application of Concurrency to System Design (ACSD 2003), June 2003, Guimaraes, Portugal*, pages 41–50, 2003.

[6] S. Bouveret, J. Brunel, D. Chemouil, and F. Dagnaty. Towards a categorical framework to ensure correct software evolutions. In *Proc. 27th International Conference on Data Engineering Workshops*, ICDEW '11, pages 139–144, Washington, DC, USA, 2011. IEEE.

[7] S. Chaki, N. Sharygina, and N. Sinha. Verification of evolving software. In *Proc. 3rd Workshop on specification and verification of component based systems (SAVCBS)*, pages 55–61, Oct. 2004.

[8] R. P. Cook and I. Lee. Dymos: A dynamic modification system. In *Proc. Software engineering symposium on High-level debugging*, SIGSOFT '83, pages 201–202, New York, NY, USA, 1983. ACM.

[9] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *Formal Methods in System Design*, volume 19, pages 45–80. Kluwer Academic Publishers, 2001.

[10] A. David, G. Behrmann, P. Bulychev, J. Byg, T. Chatain, K. G. Larsen, P. Pettersson, J. I. Rasmussen, J. Srba, W. Yi, K. Y. Joergensen, D. Lime, M. Magnin, O. H. Roux, and L.-M. Traonouez. Tools for model-checking timed systems. In O. H. Roux and C. Jard, editors, *Communicating Embedded Systems – Software and Design*, pages 165–225. ISTE Publishing / John Wiley, Oct. 2009.

[11] S. Fickas and M. Feather. Requirements monitoring in dynamic environments. In *Proc. 2nd Intl. Symp. on Requirements Engineering*, pages 140–147, Mar. 1995.

[12] J. Fisher, T. A. Henzinger, D. Nickovic, N. Piterman, A. V. Singh, and M. Y. Vardi. Dynamic reactive modules. In *Proc. 22nd Intl Conf. on Concurrency Theory*, CONCUR'11, pages 404–418, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] C. Ghezzi, J. Greenyer, and V. Panzica La Manna. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *Proceeding of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012)*, 2012.

[14] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *Proc. 12th Intl. Symp. on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 179–188, New York, NY, USA, 2004. ACM.

[15] J. Greenyer. *Scenario-based Design of Mechatronic Systems.* PhD thesis, University of Paderborn, Oct. 2011.

[16] D. Gupta and P. Jalote. On line software version change using state transfer between processes. *Softw. Pract. Exper.*, 23:949–964, Sept. 1993.

[17] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Software Engineering*, 22(2):120–131, Feb. 1996.

[18] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. In *Intl. Journal of Foundations of Computer Science*, volume 13:1, pages 5–51, 2002.

[19] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for uml sequence diagrams. *Software and Systems Modeling (SoSyM)*, 7(2):237–252, May 2008.

[20] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer-Verlag, August 2003.

[21] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster. Efficient systematic testing for dynamically updatable software. In *Proc. 2nd Intl. Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 9:1–9:5, New York, NY, USA, 2009. ACM.

[22] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16:1293–1306, Nov. 1990.

[23] H. Kugler, C. Plock, and A. Pnueli. Controller Synthesis from LSC Requirements. In M. Chechik and M. Wirsing, editors, *Proc. 12th Int. Conf. on Fundamental Approaches to Software Engineering, FASE 2009*, volume 5503 of *LNCS*, pages 79–93. Springer, 2009.

[24] I. Lee. *A Language and Architecture for the Dynamic Modification of Programs.* PhD thesis, University of Wisconsin - Madison, 1983.

[25] X. Liu and S. Smolka. Simple linear-time algorithms for minimal fixed points. In K. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 53–66. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055040.

[26] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In *Proc. 19th Symp. and 13th European Conf. on Foundations of software engineering*, ESEC/FSE '11, pages 245–255, New York, NY, USA, 2011. ACM.

[27] S. Maoz and D. Harel. From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In *Proc. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005, Portland, Oregon, USA*, pages 219–230, November 2006.

[28] R. Mazala. Infinite games. In E. Grädel, W. Thomas, and T. Wilke, editors, *Automata Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, pages 23–38. Springer Berlin Heidelberg, 2002.

[29] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. *Logics and models of concurrent systems*, pages 123–144, 1985.

[30] E. W. Stark. A proof technique for rely/guarantee properties. *Foundations of Software Technology and Theoretical Computer Science*, 206:369–391, 1985.

[31] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.*, 33:856–868, Dec. 2007.

[32] J. Whittle, W. Simm, and M.-A. Ferrario. On the role of the user in monitoring the environment in self-adaptive systems: a position paper. In *Proc. 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 69–74, New York, NY, USA, 2010. ACM.

[33] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proc. 28th Intl. Conf. on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.