Synthesizing Dynamically Updating Controllers from Changes in Scenario-Based Specifications

Carlo Ghezzi, Joel Greenyer, and Valerio Panzica La Manna
Dependable Evolvable Pervasive Software Engineering (DEEPSE) Group,
Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Via Golgi 42, 20133 Milano, Italy
{*ghezzi*|*greenyer*|*panzica*}@*elet.polimi.it*

***Abstract*—Many software-intensive systems are expected to run continuously while their environments change and their requirements evolve, so their implementation must be updated dynamically to satisfy changing requirements while coping with changing environment properties. Techniques for developing dynamically updating systems exist, but thus far almost no attention has been paid to defining when updates are correct with respect to a changing specification, i.e., when a system can safely disregard its current obligations and change its behavior to satisfy the new specification. Based on an intuitive example, we elaborate a formal definition for correct updates of a current implementation with respect to specification changes. Moreover, we present an approach for synthesizing a dynamically updating controller from the current implementation and changes in a scenario-based specification that updates to the new behavior as soon as possible. The presented technique is a first step towards the specification-driven development of safe dynamically updating controllers.**

***Keywords*-dynamic updates; scenario-based specification; controller synthesis**

I. Introduction

In many areas today, we find software-intensive, distributed systems that we expect to continuously operate even in changing environments and even as their requirements evolve over time. Examples range from information systems in commerce and healthcare to autonomous production and transportation systems. Often it is expensive and impractical to shut down these systems in order to perform software updates, so the software must be updated during run-time. Our goal is, in particular, to build systems that not only update *without disruption*, but also adapt to changing requirements *as soon as possible*, so that a system can quickly adapt to operate safely in critical situations.

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 [1]–[8] However, the existing approaches do not investigate the relationship between the evolving specification of the system and the updating software. Most approaches just consider that the system or its parts update in a state where they are not currently involved in any interaction [2], or where a component can be replaced if it still provides the services of the old [5]. In programming languages, there are similar approaches that require procedures to be inactive during the update [3], [9], or that an update can only be performed if the current state is also a state of the new program version [4]. However, the evolution of the software is not viewed from a requirements specification perspective and it has thus far not been considered whether an update is correct with respect to changes in the specification.

Similarly, there has been intensive research on *dynamically adaptive* systems, which are systems that dynamically change their behavior during run-time [5], [10], [11]. Languages have been proposed that typically allow for specifying software that can re-configure between a fixed set of configurations at pre-defined update points. Also, techniques exist for checking certain properties of such software. But again, to the best of our knowledge, it has not yet been investigated when a system adapts *correctly* with respect to changing requirements or environment properties.

In this paper, we address the following questions. We consider a system of interacting components that reacts to events in its environment. The system components are controlled by a finite-state controller that implements a specification consisting of *requirements* and *environment assumptions*. The requirements describe which sequences of events are allowed in the system and the environment assumptions describe which sequences of events can occur in the environment. If there is a change in the specification, we would like to know:

1) In which states of a running system is it safe to disregard the old obligations of the system and update the behavior to satisfy the changed specification?
2) Based on the old controller and the specification change, how can a controller be derived that dynamically updates to the new behavior as soon as it is safe to do so?

The contribution of this paper is twofold. First, based on an intuitive example, we elaborate a general formal definition for the states of a current controller in which a dynamic update to a behavior that satisfies the changed specification

is safely possible. Second, we present a specific approach where the requirements are formalized by a scenario-based specification and how, based on the specification changes, extensions to the current implementation of the system can be automatically synthesized such that it updates to the new behavior as soon as possible. We consider specifications in the form of Modal Sequence Diagrams (MSDs) [12], a variant of Live Sequence Charts (LSCs) [13]. MSDs allow us to formally specify which sequences of interactions may, must, or must not occur during runs of the system. Such a specification may evolve by adding or removing MSDs. The approach for synthesizing the dynamically updating controller is based on a synthesis technique for MSDs that one of the authors elaborated earlier [14].

In the scope of this paper, we consider that specification changes are specified by an engineer, but our long-term vision is that specification changes may in the future also be derived by components within the system at run-time, e.g., from high-level goals or user input [15], [16]; likewise changes in environment assumptions may also be induced 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 requirement changes and changes of environment properties.

The paper is structured as follows. Based on an intuitive example in Sect. II, we give a formal definition of updatable states and correct updates in Sect. III. We then introduce evolving scenario-based specification in Sect. IV and describe the approach for synthesizing dynamically updating controllers in Sect. V. Last, we describe related work in Sect. VI and conclude in Sect. VII.

II. Requirements Evolution Examples

As an example, we consider an evolving specification of the RailCab system[1]. The RailCab system is a concept for the future rail-bound traffic 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 that approaches 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 request the crossing control the permission to enter (`requestEnter`), which must reply whether entering the crossing is allowed or not (`enterAllowed(true/false)`) before the RailCab passes the point where for the last time it can be guaranteed that braking will safely stop the RailCab before it will enter the crossing (`lastBrake`).

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

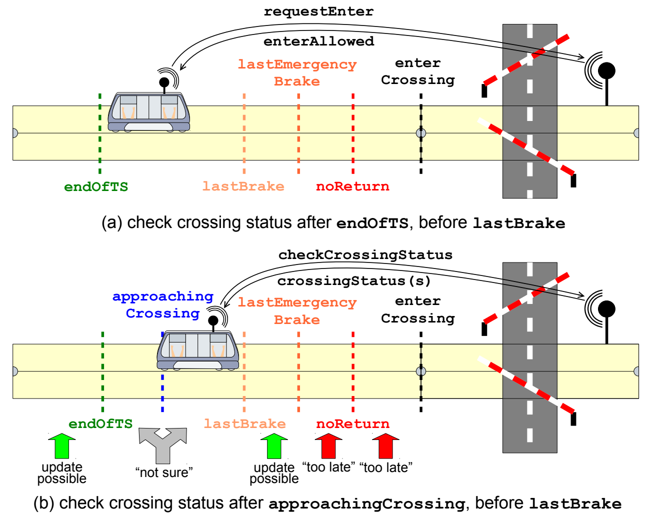

Figure 1. A RailCab approaches a crossing

The RailCab then passes two other points: `lastEmergencyBrake` and `noReturn`. In between these points, by applying the emergency brakes, the RailCab will be able to stop before entering the crossing. If the RailCab is not allowed to enter, it must brake before reaching `noReturn`, the point of no return. If it is allowed to enter, it will eventually enter the crossing (`enterCrossing`). Meanwhile, the crossing control will have closed the crossing gates.

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 gates of the crossing, 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 reason, 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 it is 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 an online update

when the controller is in its initial state would lead to an equivalent observable behavior. 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 the same behavior as an offline update or online update in the initial state. For example, after `endOfTS` has already occurred, it is still possible to update to the new requirements, because nothing has happened yet that is forbidden according to the new requirements. 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 requirements. 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, an implementation of the old requirements as described above will 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 both the old and 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.

We assume that system controllers typically visit their initial state periodically so that eventually an update will again be possible.

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.

III. Correct Dynamic Updates

We give preliminary definitions in Sect. III-A before defining updatable states and correct updates in Sect. III-B.

A. Object Systems, Controllers, Runs, Specifications

We consider systems of objects that exchange messages. For simplicity, 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. The sending and receiving of a message is 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*.

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* $\forall i \geq 1 : (q_i, m_i, q_{i+1}) \in T$*.*

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 involving objects controlled by both controllers.

Definition 3 (Parallel composition)**.** *Let* c_1 *and* c_2 *be two controllers for disjoint sets of objects. Furthermore, let their events,* Σ_1 *and* Σ_2*, only be such events where the sending or receiving object is controlled by* c_1 *resp.* c_2*. If* $\Sigma_1 \cap \Sigma_2 \neq \emptyset$*, 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 a specification.

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 run is also said to be admissible *with respect to a specification iff it satisfies this specification. 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 any event 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.*

Put another way, the environment can never block any event occurring in the system. Conversely, 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 the system objects* implements *or* realizes *S iff c composed with every possible controller e for the environment objects satisfies S, more formally $\forall e, e||c \models S$.*

B. Histories and Updatability

We now return to the problem of understanding when a dynamic update can be safely performed. In Sect. II, we intuitively argued that an online update is admissible if it leads to same observable behavior as an offline update, which involves shutting down the system and restarting the system with a new controller in its initial state.

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 be shut down and restarted with the (unchanged) controller c in its initial state without violating the specification S. We can thus imply that shutting down and restarting a system leads to an equivalent behavior as a controller visiting its initial state.

An online update of a current controller c that implements a specification S to a changed specification S' would thus be admissible if it leads to a run that 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 the changed specification S'.

An online update is thus possible if the current controller is in its initial state, but there are even 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 lead 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}, and there are different sequences of transitions leading to this state from the start state, it may be that different sequences of events may have occurred in the past. These possible sequences of events are called the *possible recent histories* of a state q_{cur}. 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'. Second, we require that there must not be any confusion on how to continue executing the system. This means that continuations that complete one possible recent history to a run that satisfies S' must also complete every other possible recent history into a run that satisfies S'.

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 whether we should already consider possibly 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* that have just not been captured thus far. The event `approaching-Crossing` in the RailCab example (see Fig. 1 (b)) could be a signal along the track section that is already detected by the RailCabs sensors, but was thus far ignored by the controller. In that case already 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 only 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 (Possible histories, possible 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* possible 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* possible recent histories *$\Pi^{<}_{past}(c, q_{c_{cur}})$ are such possible histories where the start state is not visited a second time, i.e., $\forall i \neq 0 : q_0 \neq q_i$.*

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

Definition 8 (Updatable, 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 possible 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 performs a correct update *to satisfy the changed specification S' if it behaves according to the current controller c until an updatable state is reached. Then the sequence of events that occurred since its controller 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. II 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.

IV. Evolving Scenario-based Specifications

We propose an approach where the specification of the system is given in the form of Modal Sequence Diagrams (MSDs) [12]. This section briefly explains the syntax and semantics of MSDs and we give an example of how an MSD specification may evolve.

A. 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* [14]. 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 examples formalizes the requirements described informally in Sect. II. 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`), the point of no return (`noReturn`), and will then finally enter the crossing (`enterCrossing`). For simplicity, these assumptions are overly strict: we assume that the RailCab does not brake or reverse.

B. Message Temperature and Execution Mode

The messages in a universal MSD can 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 there are cold, monitored messages shown as blue dashed arrows and hot executed messages shown as red solid arrows.

The semantics 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.

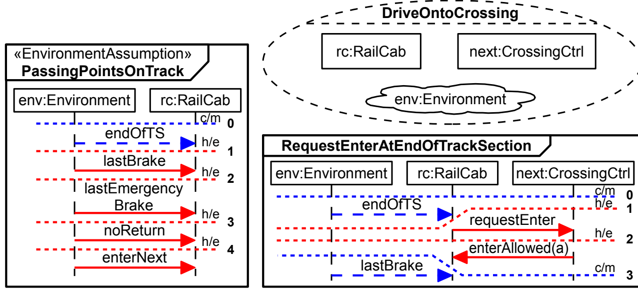

Figure 2. The example MSD specification (S)

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 an executed message is enabled, 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 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 the MSDs of Fig. 2 show the different reachable cuts. Letters indicate whether the cut is hot or cold and monitored or executed.

C. Synthesis of Controllers from MSD Specifications

Although MSDs are an intuitive formalism for precisely capturing system requirements and environment assumptions, it may be that an MSD specification is *inconsistent*, which means that there does not exist any controller that implements the specification. In order to check the consistency of an MSD specification, and to obtain a controller implementing the specification if it is consistent, we have elaborated a technique for automatically synthesizing controllers from timed and untimed MSD specifications [14]. The technique is based on mapping the synthesis problem to a two-player game that can be solved by existing, efficient algorithms that are implemented in UPPAAL TIGA [17], an extension of the UPPAAL model checker for solving two-player games. To synthesize a controller from an MSD specification, the requirement and assumption MSDs are mapped to a special variant of Timed Automata used by UPPAAL TIGA. UPPAAL TIGA can calculate a *winning strategy* for how the system objects can always react to environment events such that the specification is satisfied. The approach also allows us to calculate a *maximal* winning strategy, which for every state contains all the admissible actions of the system. Especially, we can synthesize maximal strategies where the system can also wait for environment events to occur even though there is currently an executed cut. From these winning strategies, controllers for the system can be derived. Controllers derived from maximal winning strategies are called *maximal controllers*.

We assume that the environment can always satisfy the environment assumptions regardless of the system. To check this, the synthesis technique also allows us to synthesize a controller for the environment from the environment assumptions only.

D. 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 a requirement MSD is 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 an assumption MSD is removed, it means that some assumption about the environment is no longer valid.

We consider the changed specification as explained informally in Sect. II: we replace the assumption MSD PassingPointsOnTrack by one that also contains the environment event `approachingCrossing` and 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. The changed set of assumption and requirement MSDs is shown in Fig. 3.

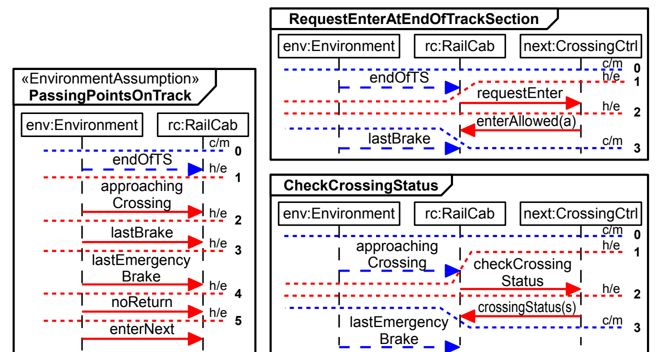

Figure 3. The changed MSD specification (S')

V. SYNTHESIZING DYNAMICALLY UPDATING CONTROLLERS

In this section we describe our approach to synthesize a *dynamically updating controller*, which behaves as the current controller and, as soon as an updatable state is reached, dynamically updates to the new behavior. As illustrated in Fig. 4, given the controller c and the MSD specifications S and S', we automatically construct a dynamically updating controller, which consists of parts of the current controller c and a new controller c', which we synthesize from the changed specifications. In updatable states of c, the according states in the c-part of the dynamically updating controller have *update transitions* to states in the c'-part, where the execution of the system is continued according to the changed specification.

Without considering the technical details on how this is done on a specific platform, we provide a procedure for *installing* the dynamically updating controller on a running system. In this procedure, all the states of the current controller are maintained, and especially the current runtime state of the current controller remains unchanged. We change the current controller as follows. During the installation procedure, we add the c'-part as well as the update transitions from the updatable states in c to the corresponding states in the c'-part. Furthermore, we remove all other outgoing transitions from updatable states in the c-part. Added states and transitions are labelled with "++" in Fig. 4; the removed transitions are crossed-out. We assume that the installation can be performed without interfering with the system's observable behavior.

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 system in any environment that behaves according to the new environment assumptions A'. This is done by synthesizing the maximal environment controller e' from A', which models any possible environment satisfying A', and then computing the parallel composition $e'||c$.
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 $e'||c'$, for which 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. We build a controller by combining c with c'. Then we transform the history relation into update transitions between states of c and c' if such a transition can be derived in a unique way. Such a unique way exists for a state q_1 in c to a state q_1' in c' if 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'. A state in the c-part where an update transition is added is an updatable state. Last, from these states, we remove all other outgoing transitions.

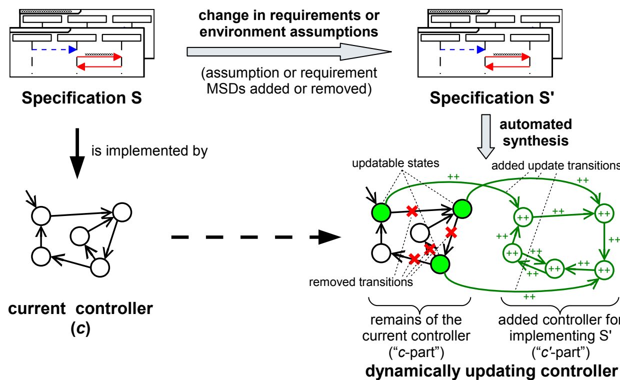

Figure 4. The approach for synthesizing a dynamically updating controller from a specification change and the current implementation of the system

In the following subsections we describe our approach by referring to the RailCab system example. Moreover, we informally justify why our approach is correct and complete.

A. 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. Furthermore, we assume that the specification was changed to the MSD specification S' in Fig. 3. 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 controller, are called *uncontrollable* transitions. They are shown as dashed arrows. Vice versa, if the sending object of the message is controlled by the controller, we call the transition a *controllable* transition. 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' cor-

Figure 5. Synthesis Approach

respond to cuts in that MSD accordingly[2].

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.

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).

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

B. Correctness of the Approach

In the following, we justify that our approach correctly and completely identifies updatable states and that it will construct a dynamically updating controller that performs correct updates according to Def. 8.

First we discuss on how our approach identifies all the updatable states. The justification is divided in four steps:

- By synthesizing the maximal controller e', and by computing the parallel composition $e'||c$ we found a way, as required in the definition, to derive all possible recent histories $\pi^{<}_{past} \in \Pi^{<}_{past}(c, q_{c_{cur}})$ for all states $q_{c_{cur}}$ in the current controller c. The definition also requires the existence of a controller c' that implements

Figure 6. Dynamically Updating Controller

[2]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.

S' and where the composition with any possible environment controller e' has a trace language $L(e'||c')$. In our approach we synthesize the maximal controller c', which includes any other controller implementing S'.

- By the way that we compute the history relation, we are able to justify the first criterion in the definition of updatable states: our approach maps a state $q_{cur} \in e'||c$ with a state $q'_{cur} \in e'||c'$, if every possible recent history of $q_{c_{cur}} \in c$ is a recent history of $q'_{c_{cur}} \in c'$. This means that for every state q'_{cur} in the history relation, there exists a run $\pi \in L(e'||c')$ passing from q'_{cur} with a prefix equals to any $\pi^{<}_{past} \in \Pi^{<}_{past}(c, q_{c_{cur}})$.
- By the way that we create update transitions and thereby identify updatable states in Step 4, also the second criterion in the definition of updatable states is satisfied: Our approach does not create an update transition from a state in the current controller from which the history relation implies different possible target states in c'. This would mean that there are different ways of continuing the execution in this state. More formally, there would be a continuation π_{future} of a recent history $\pi^{<1}_{past} \in \Pi^{<}_{past}(c, q_{c_{cur}})$ which is not a valid continuation of another recent history $\pi^{<2}_{past} \in \Pi^{<}_{past}(c, q_{c_{cur}})$ with $\pi^{<2}_{past} \neq \pi^{<1}_{past}$. Because such states are not marked as updatable states, our approach only identifies states that fulfill also this second criterion.
- Our approach is complete and identifies *all* updatable states. Since the synthesized controllers e' and c' are maximal, it means they include all possible sequences of environment and system events. Therefore, by using any other environment controller that satisfies A' or any other system controller implementing S', we would not be able to find any other updatable state.

After installing the dynamically updating controller, the running system performs a correct update for the following reasons. Consider that the running instance of the system is in state $q_{c_{cur}}$. The state $q_{c_{cur}}$ may be an updatable state or not. If $q_{c_{cur}}$ is an updatable state, it implies by Def. 8 that the recent history that occurred since c visited its initial state for the last time can be completed to a run that satisfies S'. It will be complete to a run satisfying S' because the all the old transitions from $q_{c_{cur}}$ were removed and the only remaining transition is the update transition, that takes the dynamically updating controller to a corresponding state $q_{c'_{cur}}$ in the c'-part. This state has the same recent history and from here, this history will be completed to a run satisfying S'.

If $q_{c_{cur}}$ is not an updatable state, this means that all the old outgoing transitions of $q_{c_{cur}}$ are kept and the controller will behave like c until it reaches an updatable state.

VI. Related Work

Dynamic software updates have been studied in the past. The problem has been addressed in the area of programming languages [1], [3], [4] and from the perspective of distributed, reactive systems [2], [5], [6], [18], [19].

The early work on dynamic software updates in the area of programming languages required that procedures affected by the changes were currently idle [1], [3]. Later, Gupta et al. [4] 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 the 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." [4, p. 122]. Our motivation is similar, but we 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 [2], [18], [19], 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 [5]. This, however, 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 [10], [11]. The languages they propose allow for specifying software that can reconfigure between a fixed set of configurations at pre-defined update points. Geise et al. propose a formalism based on state charts and regard mainly the reconfiguration of continuous controllers [10]. Zhang et al. propose a formalism for modeling adaptive software that requires the manual definition of update points [11]. They provide a specification language and verification support for temporal properties that are invariant during the adaptation or adaptation-specific. Both approaches, however, 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. [8] 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 allowable update points is not considered, nor is any relation defined between update tests and specifications changes.

To the best of our knowledge, this paper proposes the first approach for automatically synthesizing a dynamically updating controller from specification changes.

VII. Conclusion and Outlook

In this paper, we considered the question of when a dynamic update of a controller is correct with respect to changes in its specification. Based on an intuitive example,

we elaborated a definition for updatable states and for correct updates. Furthermore, we developed an approach for automatically synthesizing a dynamically updating controller based on a current controller and changes in a behavioral specification. We considered the specification to be formalized by MSDs. However, the approach can also be adapted to other specification formalisms like linear temporal logic or automata on infinite words.

This paper takes a specification-oriented perspective on dynamic software updates. This is crucial because changes in practice are mostly considered on the specification level first. Moreover, we present an automated approach for making safe updates to critical systems more quickly available. We also envision to build 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. Here, our approach is the key technique for automatically computing safe dynamic adaptations.

In the future, we plan to implement and evaluate our approach. It may be that, depending on the kinds of specification changes, weaker or stricter criteria for updatable states could be required. Also, we would like to extend the approach to support the synthesis of a distributed dynamically updating controller for each system object. This possible in principle [20], but rigorous techniques for doing so are still subject to current research. Furthermore, since time is an important aspect in many critical systems, we also plan to address real-time scenario-based specifications, which are, however, already supported by our synthesis technique [14].

Another issue that will be critical for the practical applicability of our technique is the inherent complexity of the synthesis problem. However, since specification changes are mostly just evolutionary, the synthesis of a controller for the new system can be greatly aided by looking at what were admissible actions in the old controller. Such optimizations pose an interesting new research question.

Acknowledgment

This research is funded by the European Commission, Programme IDEAS-ERC, Project 227977 SMScom.

References

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

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

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

[4] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Trans. Software Engineering*, vol. 22, no. 2, pp. 120–131, Feb. 1996.

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

[6] S. Ajmani, "Automatic software upgrades for distributed systems," Ph.D. dissertation, MIT, Cambridge, MA, USA, 2004.

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

[8] 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*, ser. HotSWUp '09. New York, NY, USA: ACM, 2009, pp. 9:1–9:5.

[9] I. Lee, "A language and architecture for the dynamic modification of programs," Ph.D. dissertation, University of Wisconsin - Madison, 1983.

[10] 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*, ser. SIGSOFT '04/FSE-12. New York, NY, USA: ACM, 2004, pp. 179–188.

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

[12] D. Harel and S. Maoz, "Assert and negate revisited: Modal semantics for uml sequence diagrams," *Software and Systems Modeling (SoSyM)*, vol. 7, no. 2, pp. 237–252, May 2008.

[13] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," in *Formal Methods in System Design*, vol. 19. Kluwer Academic Publishers, 2001, pp. 45–80.

[14] J. Greenyer, "Scenario-based design of mechatronic systems," Ph.D. dissertation, University of Paderborn, Oct. 2011.

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

[16] 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*, ser. SEAMS '10. New York, NY, USA: ACM, 2010, pp. 69–74.

[17] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "UPPAAL-Tiga: Timed Games for Everyone," in *Proc. 18th Nordic Workshop on Programming Theory (NWPT'06), Reykjavik, Iceland*, L. Aceto and A. Ingolfdottir, Eds. Reykjavik University, 2006.

[18] 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.*, vol. 33, pp. 856–868, Dec. 2007.

[19] 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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 245–255.

[20] D. Harel and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications," in *Intl. Journal of Foundations of Computer Science*, vol. 13:1, 2002, pp. 5–51.