Formalizing Correctness Criteria of Dynamic Updates Derived from Specification Changes

Valerio Panzica La Manna*, Joel Greenyer*, Carlo Ghezzi* and Christian Brenner†
* Dependable Evolvable Pervasive Software Engineering (DEEPSE) Group,
Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Via Golgi 42, 20133 Milano, Italy
{panzica|greenyer|ghezzi}@elet.polimi.it
† Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn,
Zukunftsmeile 1, 33102 Paderborn, Germany
cbr@uni-paderborn.de

***Abstract*—Modern software-intensive systems often have to be updated to adapt to unpredicted changes in their environments or to satisfy unpredicted requirement changes. Many systems, however, cannot be easily shut down or are expected to run continuously. Therefore, they must be updated dynamically, at run-time. Especially for critical systems, dynamic updates must be safe and performed as soon as possible. We recently studied the relationship between specification changes and dynamic updates and defined a criterion for when a system can safely disregard its current obligations and how it should change its behavior to satisfy the new specification. In this paper, we study further examples that show that stronger and weaker variants of our original criterion are relevant when engineering dynamically updating software. We formalize these criteria and discuss their safety. Moreover, we provide a tool for synthesizing dynamically updating controllers from changes in scenario-based specifications that respect the new criteria.**

***Index Terms*—dynamic updates; scenario-based specification; update criteria;**

I. Introduction

Modern software systems are subject to continuous, often unanticipated changes. Changes may occur in the environment in which the system is operating, or in the requirements, when new functionality is added or the existing functionality is modified. To incorporate these changes, the system is typically updated *offline*, which means shutting it down, updating, and restarting it. However, many of these systems, from financial information systems to autonomous vehicles, are required to run continuously and cannot be shut down at any time. Therefore, their implementation must be updated *dynamically*, at run-time. Especially in critical applications, updates must be *safe* and, moreover, we wish to perform updates *as soon as possible*, so that the system can quickly adapt to environment changes or can incorporate new, critical requirements.

We consider open reactive systems where a system in an uncontrollable environment is controlled by a finite-state controller. The system controller 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.

In recent work, we introduced a new specification-oriented perspective for the design of dynamically updating controllers [1] and defined a fundamental criterion for correct updates. If there is a change in the specification, this criterion defines in which state the system is *updatable*, i.e., where it can safely disregard the obligations given by the current specification and start behaving according to the new specification.

According to this criterion, dynamic updates are guaranteed to be equivalent to an offline update (in terms of the events in the specification). Under the assumption that an offline update is safe, which especially means that a system can be safely shut down and restarts in its initial state, we argue that dynamic updates satisfying this criterion are also safe.

We also elaborated a constructive technique for automatically synthesizing a dynamically updating controller from changes in formal scenario-based specifications [1], [2]. This technique was implemented within ScenarioTools, our tool suite for the design of scenario-based specifications.

In the course of our work, however, we discovered that there are specification changes for which the dynamic update behavior desired by an engineer is not allowed by our fundamental criterion or is not supported by our constructive technique.

In this paper, we introduce and formalize additional criteria for correct dynamic updates that are relevant when engineering dynamically updating systems. With the help of motivating examples, we present typical kinds of specification changes for which these criteria guide the engineer in designing the desired update behavior.

First (1), we show that there are cases where dynamic updates lead to different behaviors while still corresponding to an offline update, therefore being safe. States where different update behaviors are possible we call *poly-updatable*, and an engineer could choose the adequate update behavior.

Second (2), we define the criterion of *weak updates* that identifies a larger number of updatable states. This criterion allows for more timely dynamic updates where otherwise, according to our fundamental criterion, they would only be allowed much later.

Third (3), we show that a system can be in a cycle where, according to our fundamental criterion, we cannot guarantee

that it will ever reach an updatable state. We therefore introduce the criterion of *cycle-agnostic updates*, that introduces additional opportunities for performing updates when the new specification changes this cyclic behavior.

We show that weak and cycle-agnostic updates are not equivalent to an offline update. The emerging behavior may therefore in some cases not be safe, but the criteria are sufficiently restrictive to rule out unsafe dynamic updates in many cases. Still, the design process should include a subsequent validation step, which we sketch as an outlook.

Finally, we extend our constructive technique for creating dynamically updating controllers to consider the new criteria. This allows us to automatically synthesize controllers that exhibit the desired update behavior for typical kinds of specification changes. We prototypically implemented the extended approach in ScenarioTools.

The paper is structured as follows: Sect. II provides the foundations and summarizes our fundamental criterion for updatable states and correct updates. We introduce and formalize poly-updatable states in Sect. III, weakly updates in Sect. IV, and cycle-agnostic updates in Sect. V. We overview our constructive synthesis technique and its implementation in Sect. VI. In Sect. VII we discuss related work and conclude in Sect. VIII.

II. Foundations

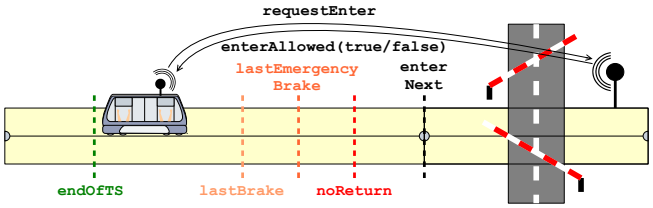
We recently introduced and formally defined a criterion for safe dynamic updates with respect to changes in the specification [1]. We provide a summary of this work as the foundations of this paper. Based on an example, we first present the intuitive idea of safe dynamic updates. Then we report the formal definitions of our previous work that will be used throughout the paper.

A. Evolving Specifications and Safe Dynamic Updates

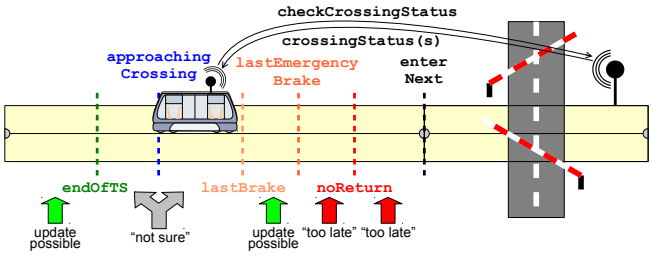
In our previous work, we considered an evolving specification of the RailCab system[1]. In the RailCab system autonomous vehicles, called *RailCabs*, transport goods and passengers on demand.

As an example, we considered a scenario of a RailCab approaching a crossing, see Fig. 1(a). When approaching the crossing, the RailCab observes certain environments events in a certain order, which correspond to certain points it passes on the track. The scenario starts when the RailCab detects that it approaches the end of the current track section (`endOfTS`). Next, it passes the points `lastBrake`, `lastEmergencyBrake`, `noReturn`, and `enterNext`. The event `enterNext` represents the RailCab entering the crossing, and `noReturn` is the point beyond which the RailCab cannot be stopped anymore from entering the crossing. Between `lastEmergencyBrake` and `noReturn`, the RailCab can only by applying the emergency brakes come to a stop before the crossing. Between `lastBrake` and `lastEmergencyBrake` is the last time when the RailCab can avoid entering the crossing by a normal braking procedure.

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



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



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

Fig. 1. A RailCab approaches a crossing

We consider that the system is currently running, realizing the following requirements. After `endOfTS` the RailCab must request the crossing control the permission to enter (`requestEnter`), which must reply whether entering the crossing is allowed or not (`enterAllowed(true/false)`). The reply must be sent before `lastBrake` so that the RailCab can brake normally in the case that entering the crossing is not allowed. For simplicity, we assume that the RailCab and the crossing control are controlled by a global controller that implements only the described scenario; this controller is in its initial state before `endOfTS`.

Now suppose that it was observed that in the case of a power outage the RailCab may enter a crossing while the crossing control could not shut the barriers, thus increasing the risk of accidents. To avoid this situation, the requirements are changed. 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`. To trigger this interaction later, another signal was installed on the track, called `approachingCrossing`. The RailCab will pass this point after `endOfTS` and before `lastBrake`. Figure 1(b) illustrates the additional requirement and the additional environment event.

To fulfill the new requirements, we can perform an offline update. This requires first to shut down the system, to install the updated controller, and to restart it in its initial state. However, this cannot be done while the RailCab is approaching the crossing. Obviously, shutting down the RailCab while it is moving would not be safe. But even if this was not a

problem, within the scenario, shutting down the system would mean that the system would forget critical obligations, for example having to send the `requestEnter` message after `endOfTS`. Therefore, we must continue running the system with its current implementation until it completes the scenario of passing the crossing. An offline update can thus *not prevent* an accident in the case of a power outage on the *current* crossing.

To avoid deadly accidents, instead we would like to update the RailCab to the new behavior dynamically, at run-time. Moreover we would like to update the system as soon as possible, even and especially if a RailCab is currently approaching a crossing.

We define *updatable states* to be states where the system can disregard the current specification and continue behaving according to the new specification in such a way that we can ensure a behavior that is equivalent to an offline update (in terms of the observable events). The arrows at the bottom of Fig. 1(b) illustrate which states in the scenario are or are not updatable.

Performing a dynamic update before `endOfTS` is possible because, obviously, changing the implementation when the controller is in its initial state, before the scenario starts, corresponds to an offline update. After `endOfTS` and before `lastBrake`, no dynamic update can be performed. The problem is that the specification implemented by the current controller does not contain the event `approachingCrossing`. Therefore we must conclude that the current controller is not aware of this event. This make it impossible to behave equivalently to an offline update; if we assume that `approachingCrossing` did not yet occur while instead it did, the RailCab will never send `checkCrossingStatus`.

After `lastBrake` occurred, and because we know that `approachingCrossing` must have happened before that, the RailCab is again in an updatable state. We can update to a controller that implements the new behavior, into a state where `checkCrossingStatus` is sent next. After `lastEmergencyBrake` occurred, there is no updatable states until the end of the scenario, because the RailCab should have checked the crossing status before.

B. Correct Dynamic Updates

We give preliminary definitions in Sect. II-B1 before defining updatable states and correct updates in Sect. II-B2.

1) Object Systems, Controllers, Runs, Specifications: We consider systems of *objects* that exchange messages as defined by Harel and Marelly [3]. 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, $\mathcal{L}(c) \subseteq \Sigma^\omega$ is the* trace language *of c. A run $\pi = (m_0, m_1, \ldots)$ is an element of $\mathcal{L}(c)$ 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. The parallel composition is defined in the usual way.

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 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 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 $\mathcal{L}(c)$ 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. Conversely, the system can perform any finite number of steps before the next environment event occurs[2].

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.

2) Histories and Updatability: Our definition of updatable states is based on the notion of the *recent histories* of a state, which are all the possible sequences of events leading to a state of a controller since it last visited the initial state.

The definition of updatable states requires the following two conditions. Intuitively, first, it must be possible to complete all recent histories to a run which satisfies the new specification. How a recent history is completed depends on what happens in the environment, so a "completion" is a set of many possible sequences of events. To reflect this, the definition requires the existence of a controller implementing the new specification and where all the recent histories of updatable states are prefixes of runs of this controller.

The second condition requires that there must not be any confusion on how to continue the execution, which means that a continuation of one recent history must also be a possible continuation for any other recent history.

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. Without loss of generality, we assume that changes in the environment assumptions reflect *new insights* in how the environment is *already behaving right now*. We therefore determine the possible recent histories based on any environment e' that implements the changed environment assumptions A'. [3]

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 $\mathcal{L}(e'||c) \subseteq A'$. 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 $\mathcal{L}(e'||c')$ where*

1) *for every recent history $\pi^{<}_{past} \in \Pi^{<}_{past}(c, q_{c_{cur}})$ there must be a run $\pi \in \mathcal{L}(e'||c')$ 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, $\forall \pi^{<1}_{past}, \pi^{<2}_{past} \in \Pi^{<}_{past}(c, q_{c_{cur}}), \pi^{<1}_{past} \neq \pi^{<2}_{past} : \pi^{<1}_{past} \cdot \pi_{future} \in \mathcal{L}(e'||c') \Rightarrow \pi^{<2}_{past} \cdot \pi_{future} \in \mathcal{L}(e'||c')$*

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'.*

Depending on the requirements change, there can be more or less updatable states. Our definition of updatable state implies that the initial state of a system is always updatable. To guarantee that a dynamic update will eventually take place, we assume that system controllers typically visit their initial state periodically so that eventually an update will always be possible.

III. Longer Histories and Poly-updatable States

Let us consider a new example of a possible specification change for the RailCab system. A current specification again describes what shall happen when a RailCab approaches a crossing, see the top of Fig. 2. Now, additionally, the engineers would like the RailCab to regularly perform a procedure for autonomously checking the wear-off of its wheels by using on-board scanners. Such a procedure can be costly, because the RailCab needs to slow down to a steady moderate speed for scanning the surface of the wheels. For this reason, imagine that the engineers decide to perform the scanning procedure when approaching and before effectively entering every third crossing. The lower part of Fig. 2 illustrates the checking procedure by a self-message `checkWheels` on the RailCab that shall take place between `endOfTS` and `enterNext` before the third crossing.

[2]Def. 5 is not essential for the definition of our criteria, but adopted in our approach [1] and our tool. We could for example also consider a setting where the environment always has priority over the system, but this would then require us to formulate more assumptions about the environment explicitly.

[3]If the new environment assumptions describe changes in the environment behavior that will only occur during or after the update, then the old environment assumptions must be considered to determine the possible recent histories.

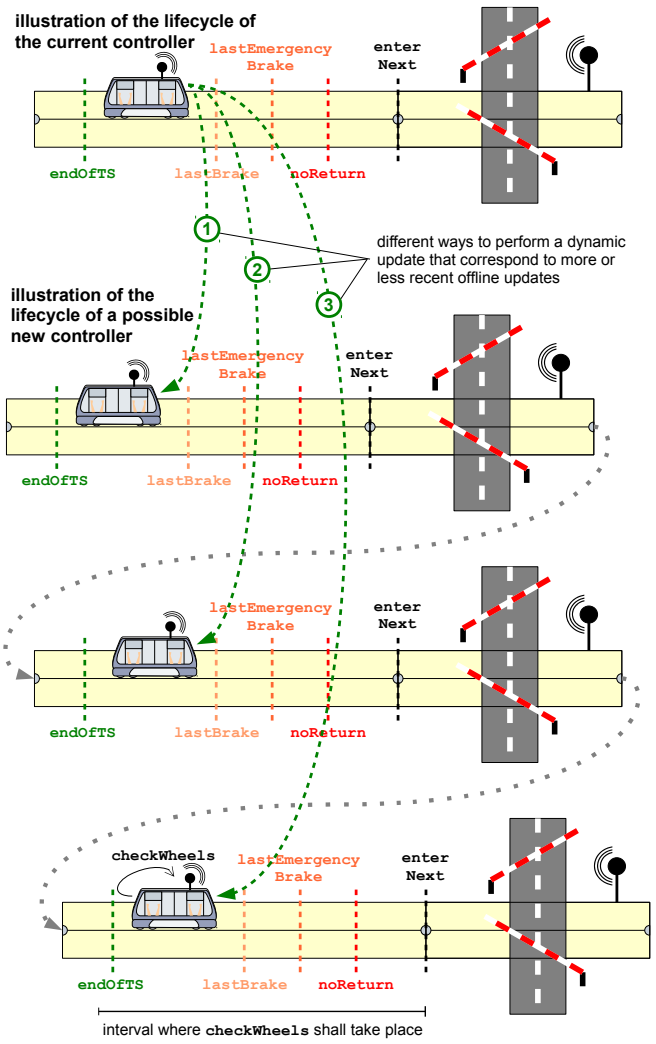


Fig. 2. The RailCab autonomously checking its wheels' wear-off when approaching every third crossing

Assuming that a current controller does not remember or count the number of crossings it already approached, the recent histories of any state of the current controller will always reflect that the RailCab just approaches its first crossing. Based on this notion of recent histories, all states of the current controller would be updatable and, if the RailCab approaches a crossing (between `endOfTS` and `enterNext`), we could update the behavior so that the wheels-checking procedure can take place when approaching the third crossing after the dynamic update; this is illustrated by the dashed arrow labeled "1" in Fig. 2.

However, the recent histories reflect the notion of a "most recent" corresponding offline update. Could we not assume that the RailCab already approached another crossing or two other crossings before? This could then lead to the dynamic update behavior illustrated by the dashed arrows labeled "2" and "3" in Fig. 2.

Intuitively, all these alternatives correspond to offline updates, but an engineer may for example wish to decide that the wheels-checking procedure is critical and shall take place as soon as possible. In this case the engineer would like to choose the update alternative "3". If, however, the procedure is costly and not urgent, the engineer may want to choose update alternative "1".

We call states from where different updates are possible with respect to more or less recent histories *poly-updatable* states. They are defined more formally by extending Def. 8 with the notion of *level-x histories*, which are histories that visit the initial state of the current controller x-many times.

Definition 9 (Level-x histories)**.** *For $x \in \{1, \ldots, n\}$ and a state q_{cur} of a controller c, $\Pi^{x}_{past}(c, q_{cur})$ denotes the set of histories (see Def. 7) that visit the initial state of c x-many times.*

The level-1 histories are the histories that visit the initial state only once, at the beginning, and are thus equivalent to the recent histories. The level-2 histories are all histories that visit the initial state one more time, etc. To define poly-updatable states, we first generalize (by a slight change) the definition of updatable states to define what it means for a state to be *updatable with respect to a particular subset of histories*.

Definition 10 (Updatable state w.r.t. a subset of histories)**.** *A state q_{cur} of a system controller c is* updatable with respect to a given subset of histories $\Pi^{\subseteq}_{past}(c, q_{cur}) \subseteq \Pi_{past}(c, q_{cur})$ *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 $\mathcal{L}(e'||c')$ where*

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

We now define poly-updatable states as follows.

Definition 11 (Poly-updatable state)**.** *A state q_{cur} of a system controller c is* poly-updatable *to a specification S' if it is updatable to specification S' w.r.t. the level-1 histories of q_{cur} and if it is updatable to specification S' w.r.t. at least one set of level-x histories, $x \in \{2, \ldots, n\}$.*

In other words, a state is poly-updatable if it is updatable in the original sense, and if it is furthermore updatable to some other level-x history with $x \in 2, \ldots, n$. The definition of a *correct update* in Def. 8 can remain unchanged. In the example, the illustrated state is poly-updatable, because it is level-1, -2, and -3-updatable.

If we report to the engineer that a state is poly-updatable, it may not be easy for the engineer to choose which alternative is adequate. If the requirements and the current implementation are more complex, an engineer may desire to simulate the

different behaviors resulting from different choices. We are planning to extend ScenarioTools in the future in order to support such simulations, but do not consider this in the scope of this paper.

IV. Shorter Histories and Weakly Updatable States

Suppose that now, to increase the safety of our system, we change the requirement to check the wheels' wear-off before each third crossing to the requirement to check the wheels before *every* crossing. Figure 3 illustrates how, according to our original definition of updatable states (Def. 8), the current RailCab controller would be updatable in a state where it did not yet enter the first crossing. By "first" we again refer to the fact that the current controller does not remember how many crossings it really approached before, but just counts to three and then starts over again. After the first occurrence of `enterNext`, according to our initial definition, it is too late for an update, because we did not do the `checkWheels` as would have been required by the new specification.

Intuitively, however, why should we not perform the update when the current controller approaches the second crossing? If checking the wheels is critical, shouldn't we better do it now than wait until the next crossing?

To support timely updates in such cases, we propose another, weaker criterion for updatable states. The basic idea of our new criterion is to find the same "pattern" of states that are updatable in the sense of Def. 8 also in other parts of the current controller and allow dynamic updates from these states under certain conditions.

More formally, this works as follows. First, we determine which states in the current controller are updatable in the original sense of Def. 8. Then we check if there exist other states in the controller that are "similar" to the initial state and the updatable states. We call these states *co-initial* and *co-updatable* states. They must be similar in the sense that for every co-updatable state we require that there exists a transition sequence from the co-initial state to the co-updatable state accepting every possible recent history of the corresponding updatable state.

Definition 12 (Co-initial state, co-updatable state, shorter histories)**.** *A state $q_{co-init}$ of a system controller $c = (\Sigma_c, Q_c, q_{c_0}, T_c)$ is* co-initial *with respect to a new specification S' iff there exists a non-empty set of states $Q_{update} \subseteq Q_c$ updatable to S' and for every $q_{update} \in Q_{update}$ there exists a* co-updatable *state $q_{co-update} \in Q_c$ such that $\Pi^{<}_{past}(c, q_{update}) \subseteq \Pi^{q_{co-update}}_{q_{co-init}}$, where, $\Pi^{q_{co-update}}_{q_{co-init}} = \{\pi \in \Sigma^* | (q_i, m_i, q_{i+1}), \ldots, (q_j, m_j, q_{j+1}) \in T^*_c, q_i = q_{co-init}, q_{j+1} = q_{co-update}, m_i, \ldots, m_j = \pi\}$. I.e., the recent histories of q_{update} are a subset of the event sequences accepted by all possible transition sequences between $q_{co-init}$ and $q_{co-update}$. We denote the set of co-updatable states for a co-initial state $q_{co-init}$ with $Q^{q_{co-update}}_{q_{co-init}}$. We also call $\Pi^{q_{co-update}}_{q_{co-init}}$ the* shorter histories *of $q_{co-update}$ with respect to its corresponding co-initial state $q_{co-init}$.*

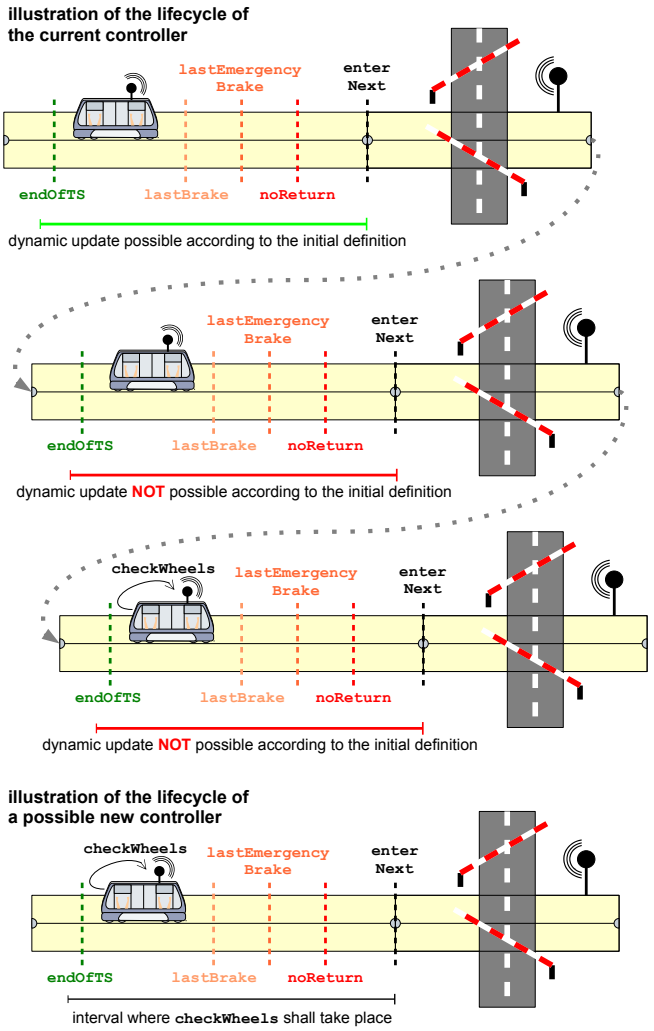


Fig. 3. Instead of checking the wheels' wear-off when approaching every third crossing, it should now happen before every crossing

We call the co-updatable states *weakly updatable* if they are updatable with respect to their shorter histories (see Def. 10).

Definition 13 (Weakly updatable state, correct weak update)**.** *A co-updatable state $q_{co-update} \in Q^{q_{co-update}}_{q_{co-init}}$ for a co-initial state $q_{co-init}$ of a current controller c and with respect to a specification S' is* weakly updatable *iff it is updatable to S' w.r.t. the shorter histories $\Pi^{q_{co-update}}_{q_{co-init}}$.*

We finally define *correct weak updates* as follows.

Definition 14 (Correct weak update)**.** *A system with controller c in a co-updatable state $q_{co-update}$ performs a* correct weak update *to satisfy the changed specification S' iff the sequence of events that occurred since c was in the co-initial state corresponding to $q_{co-update}$ for the last time will be completed to a run that satisfies S'.*

In the example shown in Fig. 3, we could find the same "pattern" of updatable states for example also if the RailCab is in front of the second crossing and we can perform a

weak update here. Before the third crossing, the system would also be weakly updatable because in this scope the system is already doing exactly what is required by the new specification. If `checkWheels` was not allowed in the new specification, the system would only be weakly updatable before `checkWheels` occurred.

While in our previous criteria for updatable states and correct updates we always guarantee that the resulting dynamic update behavior corresponds to some possible offline update, this is not true anymore for this more relaxed criterion. In fact, there are cases of correct weak updates that are *unsafe*. As the following intuitive example illustrates, we may create a dynamic update behavior where the old controller is abandoned in a state where critical obligations remain that are then not fulfilled by the new controller.

Consider a controller for the cooling system of a nuclear power plant that for a maintenance procedure stops and restarts a pump for the cooling agent. The current specification says that it is critical that whenever the pump is stopped, it must also be restarted again. In the new specification a new maintenance procedure may be adopted, which may not require starting and stopping the pump anymore. But still, we keep the requirement to always restart a stopped pump. Now imagine further that we identified some updatable and co-updatable states in the current controller where some of these are located before the pump is stopped, but some are located after the pump has stopped but before the pump was restarted; for these states, the shorter histories do not capture the fact that the pump was stopped. If we update from one of these states to a new controller that will not restart the pump, this may result in a devastating accident.

For this reason, we recommend that the design of dynamically updating controllers based on the weak update criterion includes a subsequent validation of the controllers. We envision such a validation to be supported by a tool that automatically generates traces of potentially unsafe dynamic updates from a dynamically updating controller and allows the engineer to determine whether the possible update is safe or should not be applied. The decision of an expert is required, since there are no formal properties that we could verify here that, if they were relevant, would not be included in the specification. Potentially unsafe update traces are such traces where the dynamically updating controller violates properties in the current or new specification, but especially where it violates those that remain invariant during the specification change. These traces can be generated for example using model-checking techniques, as used for test case generation [4], [5]. The details of such a technique, however, remain an outlook of this paper.

V. Cycle-free histories and cycle-agnostic updatable states

For our fundamental criterion, to guarantee a system to be always eventually updatable, we assumed a system with a repetitive behavior by which it periodically visits its initial

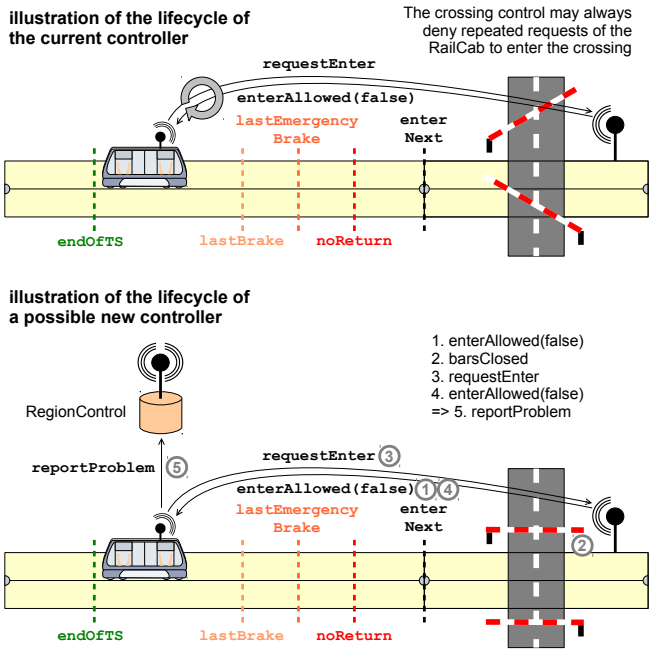


Fig. 4. New requirement: If after the bars were closed the RailCab is still not allowed to enter the crossing, it should report a problem

state. However, there are also systems with a repetitive behavior that does not necessarily lead the system into its initial state. For example, there are systems with a periodic behavior with an initialization procedure that is only executed once. Consider that now we change the specification to change or add certain steps within this repetitive behavior. According to our fundamental criterion, this is in general not possible because this criterion requires to consider previous iterations in this periodic behavior, which did not yet incorporate the changes, and will thus violate the new specification. Even worse, it may not be guaranteed that the system will eventually reach an updatable state again.

As an example consider the following case. Suppose that the current controller implements a requirement saying that if a request to enter a crossing is denied, then it should, maybe after some delay, request the permission to enter again. If for some reason the crossing control always denies these requests, it may lead to a cyclic behavior where the current controller does not visit the initial state again. The top of Fig. 4 illustrates this. At the bottom, the figure shows that now the requirements are changed. They additionally require to report a problem to a *region control*, a component to supervise regions in the RailCab track system, if a previous request to enter the crossing was denied, then the bars of the crossing closed, and then a repeated request to enter the crossing is denied again.

The problem in this case is that our previous notion of recent histories will indeed only consider such histories that visit the initial state only once, but will, in the presence of cycles not involving the initial state, contain all event sequences that correspond to looping in these cycles arbitrarily often. As a consequence, if the RailCab in the above example is in a

state where it already received an `enterAllowed(false)` message, the recent histories will contain traces where this has already happened an arbitrary number of times. If then the current controller does not observe whether the `barsClosed` events already happened or not, or it does observe this event, but does not send `reportProblem`, many possible recent histories will already violate the new requirement. Therefore, dynamic updates in the sense of our fundamental notion (Def. 8) will not be possible, and may never again be possible unless we assume that eventually the RailCab will leave this cycle and eventually visits its initial state again.

The criterion for weak updates is not tailored for this kind of specification change, because it would requires to find co-initial and co-updatable states that form a "pattern" similar to the initial state and the updatable states within the cycle. In this example, for instance, no such pattern exists.

We therefore introduce the criterion of *cycle-agnostic updates*, which describes how to achieve the desired dynamic update behavior for these kinds of specification changes. The criterion is based on the notion of cycle-free recent histories, which capture all the possible past sequences of events since the system last visited its initial state and did not yet iterate through any cycles.

Definition 15 (Cycle-free recent histories)**.** *For a state q_{cur} of a controller $c = (\Sigma_c, Q_c, q_{c_0}, T_c)$ the set of* cycle free recent histories *of q_{cur}, denoted by $\Pi^{<CF}_{past}(c, q_{cur})$, is a subset of the recent histories of q_{cur}, $\Pi^{<CF}_{past}(c, q_{cur}) \subseteq \Pi^{<}_{past}(c, q_{cur})$, for which there exists a corresponding transition sequence in c from q_{c_0} to q_{cur} that visits any state at most once.*

Our definition of cycle-agnostic updatable states is again based on Def. 10 with respect to the cycle-free recent histories.

Definition 16 (Cycle-agnostic updatable state)**.** *A state q_{cur} of a system controller c is* cycle-agnostic updatable *to a specification S' iff it is updatable to S' w.r.t. the cycle-free histories $\Pi^{<CF}_{past}(c, q_{cur})$.*

The definition of a correct cycle-agnostic update is as follows.

Definition 17 (correct cycle-agnostic update)**.** *A system with controller c in a cycle-agnostic updatable state q_{update} performs a* correct cycle-agnostic update *to satisfy the changed specification S' iff any cycle-free recent history of q_{cur} will be completed to a run that satisfies S'.*

Back to our example in Fig. 4. If we assume that the system is in a state of the cycle after it receives `enterAllowed(false)`, the cycle-free recent histories only consider that the RailCab received `enterAllowed(false)` for the first time, even though it may have iterated though the cycle many times. According to the criterion of cycle-agnostic updates, we can now perform the update and introduce the report problem mechanism into the loop as desired.

Since the considered cycle-free recent histories are only a subset of all the recent histories, the criterion of cycle-agnostic update is a relaxation of our fundamental criterion. For some recent histories the dynamic update behavior may not correspond to an offline update and can therefore not guaranteed to be safe. As we discussed in Sect. IV for weak updates, the application of cycle-agnostic updates requires additional validation support to guide the engineer in ensuring the safety of the desired update behavior.

VI. Synthesis Tool

We elaborated a constructive technique for automatically synthesizing dynamically updating controllers from a given current controller and a specification change [1]. Recently, we implemented this technique as an extension of ScenarioTools, our novel, Eclipse-based design, simulation, and synthesis tool suite for scenario-based specifications [2]. In the scope of this paper, we extended the constructive approach and our tool to the new criteria. Our tool currently supports finding poly-updatable states and the corresponding alternative update behaviors. Furthermore, it supports the synthesis of a dynamically updating controller based on the criterion for cycle-agnostic updates. The support for weak updates is still under development. The tool and examples can be downloaded from our website [4].

In our constructive technique, we consider specifications in the form of Modal Sequence Diagrams (MSDs) [6], a variant of Live Sequence Charts (LSCs) [7]. MSDs allow us to formally specify which sequences of events may, must, or must not occur in the running system. An MSD specification evolves by adding or removing MSDs.

In summary, the constructive technique works as follows. For more information, we refer to our technical report [2]. Given the controller c implementing the current specification S and the new specification S', we first synthesize a new controller c' implementing S'. To do this, we implemented an efficient incremental algorithm which synthesizes the new controller c' on the basis of the current implementation. The incremental synthesis is guided by the actions by which the current controller could successfully satisfy the current specification. If the specification change is evolutionary, the incremental synthesis can be significantly more efficient that a non-incremental synthesis. Moreover, it synthesizes a controller for the new specification that is similar to the current controller so that, as described below, we can in many cases identify all updatable states.

We then identify the *history relation* between states in c and c'. A pair of states belongs to the history relation if every recent history of the first element in the pair is also a recent history of the second. This is done by first computing the pairs of states of the controllers c and c' which have at least one common recent history. The resulting pairs are called the *candidate pairs* of the history relation. Next, we gradually remove such pairs (q, q') where 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 c' where a transition labeled with m leads to q'. One special case is the

[4] http://www.scenariotools.org

pair formed by the initial states of the two controllers, which we never remove. Eventually all the remaining pairs form the history relation.

The dynamically updating controller can now be formed by combining the current controller c, and the new controller c'. From the history relation, we then identify all the updatable states in the current controller and derive *update transitions*, which map each of these states to a corresponding target state of the c'-part of the combined controller. The dynamically updating controller can be deployed in the running system by extending the running current controller with the update transitions and the c' controller. Then, when the current controller reaches an updatable state, its update transition is immediately taken. From then on the execution will be completed by c'-part of the dynamically updating controller.

We extended our original approach to identify all the states in the current controller that are poly-updatable and cycle-agnostic updatable. The identification of poly-updatable states requires a generalization of the history relation which must also include the state pairs reached by the same level-x histories with $x \in 2, \ldots, n$. We then continue the exploration of possible target states in the new controller with a recent history that is equivalent to some level-x history of the current controller with $x > 1$.

For the identification of cycle-agnostic updatable states we only need to consider pairs of states (q, q'), q being a state of c and q' being a state of c', where all cycle-free recent histories of q are also recent histories of q'. To do this, we modify the computation of the history relation as follows. We first compute all the candidate pairs as explained above. Then, also as before, we gradually remove such pairs (q, q') where 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 c' where a transition labeled with m leads to q'. In this process, however, we ignore predecessors of q for which we previously determined that they can only be reached by a previous visit of q. This predecessor of q can be ignored, because it at the same time is a successor of q, and can therefore only be visited in a cycle.

VII. Related Work

Dynamic software updates have been studied in the past in different areas of research. In the area of programming languages, different approaches have been proposed to perform dynamic updates of Java applications [8]–[10] and C programs [11], [12]. These approaches, however, do not consider when a dynamic update is correct with respect to changes in the specification of a program. If program specifications are provided, our correctness criteria, which are independent from a specific programming language, could extend these approaches to reason about the correctness of updates.

Other approaches focus on identifying under which conditions a system can be correctly updated. Early works required that procedures affected by the changes to be currently idle [13], [14]. Later, Gupta et al. [15] 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. Our notion of updatable state is similar, but we consider the states of different finite state machines and system specifications and more generally argue over the sequences of events.

Recent techniques were proposed to test dynamically updating programs [16] or to verify their correctness [17]. In both these approaches the update points are specified manually. Criteria for automatically finding allowable update points are not considered, nor is any relationship between updates and specifications changes defined.

Dynamic updates were also studied in the area of dynamically reconfiguring component-based systems [18]–[22]. The criteria introduced by Kramer and Magee [18], Vandewoude et. al [19], and Ma et. al [20], enable safe dynamic reconfiguration of components. These criteria, however, may be too restrictive because the component to be updated is required to be in a state where no interactions are currently active. The updatable states resulting by our criteria, instead, allows for more timely dynamic updates. Chaki et al. define that a component can be updated if it still provides the same functionality of the old [21]. This, however, implies that the specification cannot become more restrictive. Giese et al. proposed a formalism based on state charts and regard mainly the reconfiguration of continuous controllers [22], but this approach does not consider the validity of updates with respect to specification changes.

In the area of self-adaptive systems, different approaches have been elaborated to modeling and verifying adaptive software [23]–[26], [26]. These papers propose languages for specifying software that can reconfigure between a fixed set of configurations at pre-defined update points. Zhang et al. propose a formalism for modeling and verifying adaptive software which requires the manual definition of update points [23], [24]. They provide a specification language and verification support for temporal properties that are invariant during the adaptation or adaptation-specific. In our approach, instead, the update points satisfying the introduced criteria are automatically identified on the basis of the specification change. Adler et al., propose a framework for developing dynamically adaptive embedded systems [25], Bouveret et al., describe a categorical framework to ensure correct software evolutions [26], and Fisher et al., propose a formal language for modeling adaptive software [27]. These approaches, however, do not consider that certain configurations comply to certain requirements and that reconfigurations must satisfy certain conditions with respect to these requirements.

VIII. Conclusion

In this paper, motivated by intuitive examples, we introduced and formalized novel correctness criteria for dynamic updates from specification changes. Depending on the specific needs of the application and the type of specification change, the engineer may then choose which criterion to adopt for the design of a dynamically updating controller.

The criterion of poly-updatable states allows the engineer to identify states in which there are several alternative ways

to perform safe dynamic updates, where safe means that the resulting behavior is guaranteed to be equivalent to an offline update. We also showed that more relaxed updatability criteria are required for certain kinds of specification changes in order to allow for dynamic updates that may otherwise be impossible or would only be performed much later than desired.

Moreover, we extended our constructive synthesis approach to support a subset of the newly introduced criteria, so that dynamically updating controllers with the desired update behavior can be synthesized automatically. We implemented these extensions within SCENARIOTOOLS.

The relaxed criteria may be unsafe since they do not guarantee the dynamic update to be equivalent to an offline update. Therefore, their applicability should include a subsequent validation step. We sketched how such a validation could show potentially unsafe traces to the engineer, who must then decide whether the update behavior is safe or should be excluded. The details of such a technique remain an outlook of this paper.

Our constructive synthesis technique currently synthesizes a global dynamically updating controller for the entire system. For distributed system, our future goal is to be able to synthesize distributed dynamically updating controllers that interact and update to still satisfy our correctness criteria.

Also, we believe that there are yet other updatability criteria for further kinds of specification changes. Especially, there may be specification changes that are a combination of multiple different kinds of changes for more or less interdependent behavioral aspects within a specification. We believe that how to design or automatically synthesize the desired and safe dynamically updating controllers in such cases is an interesting and practically relevant new research direction.

ACKNOWLEDGMENT

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.

REFERENCES

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

[2] J. Greenyer, V. Panzica La Manna, C. Brenner, and C. Ghezzi, "Synthesizing safe dynamic updates from evolving specifications," Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Tech. Rep. n. 2013.2, Jan. 2013. [Online]. Available: http://scenariotools.org/docs/2013/SynthesizingDynamicallyUpdatingControllers-TechRep.pdf

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

[4] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 146–162, Oct. 1999.

[5] H. Zeng, H. Miao, and J. Liu, "Specification-based test generation and optimization using model checking," in *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, June 5-8, 2007, Shanghai, China*, 2007, pp. 349–355.

[6] 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.

[7] 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.

[8] A. R. Gregersen and B. N. Jørgensen, "Dynamic update of java applicationsbalancing change flexibility vs programming transparency," *J. Softw. Maint. Evol.*, vol. 21, no. 2, pp. 81–112, Mar. 2009.

[9] J. Kabanov, "JRebel tool demo," *Electron. Notes Theor. Comput. Sci.*, vol. 264, no. 4, pp. 51–57, Feb. 2011.

[10] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lu, "Javelus: A low disruptive approach to dynamic software update," in *19th Asia-Pacific Software Engineering Conf. (APSEC 2012)*. Hong Kong, China: IEEE Computer Society, dec. 2012, pp. 527–537.

[11] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005.

[12] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," *SIGPLAN Not.*, vol. 41, no. 6, pp. 72–83, Jun. 2006.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *Proc. Intl. Conf. on Verified Software: Theories, Tools, and Experiments (VSTTE)*, Jan. 2012.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] J. Zhang, H. J. Goldsby, and B. H. Cheng, "Modular verification of dynamically adaptive systems," in *Proc. 8th Intl. Conf. on Aspect-oriented software development*, ser. AOSD '09. New York, NY, USA: ACM, 2009, pp. 161–172.

[25] 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*, ser. ICFEM'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 76–95.

[26] 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*, ser. ICDEW '11. Washington, DC, USA: IEEE, 2011, pp. 139–144.

[27] 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*, ser. CONCUR'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 404–418.