Towards Executing Dynamically Updating Finite-State Controllers on a Robot System

Valerio Panzica La Manna*, Joel Greenyer†, Donato Clun‡, and Carlo Ghezzi‡
*MIT Media Lab, Cambridge, MA, USA
Email: vpanzica@mit.edu
†Software Engineering Group, Leibniz Universität Hannover,
Welfengarten 1, 30167 Hannover, Germany
Email: greenyer@inf.uni-hannover.de
‡Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano,
Piazza Leonardo Da Vinci, 32, 20133 Milano, Italy
Email: donato.clun@mail.polimi.it carlo.ghezzi@polimi.it

***Abstract*—Modern software systems are increasingly required to run for a long time and deliver uninterrupted service. Their requirements or their environments, however, may change. Therefore, these systems must be updated dynamically, at run-time. Typical examples can be found in manufacturing, transportation, or space applications, where stopping the system to deploy updates can be difficult, costly, or simply not possible. In previous work we proposed a model-driven approach that uses automatically synthesized finite-state controllers from scenario-based assume/guarantee specifications to safely and efficiently dynamically update the system. In this paper we describe an execution infrastructure of this approach, which allows us to execute and deploy newly synthesized dynamically updating controllers on embedded devices. We present a prototype implementation in Java for Lego Mindstorms robots. This experience gained can lead to a systematic approach to implement dynamic updates in the aforementioned critical software-intensive systems.**

I. Introduction

Modern software systems are often subject to changes. Changes may occur due to evolution of the environment in which they are embedded, or in the requirements, when the currently supported functionalities need to be extended or modified. These changes are often applied by performing an *offline update*, which means shutting down the system, applying a software patch, and restarting the system. However, many systems are expected to run continuously even when they are updated. In application areas such as manufacturing or mobility, service interruptions can be costly. In space applications, for example, interruptions can be very critical or even impossible. Therefore, the software must be updated *dynamically*, at run-time. Dynamic updates must be *safe*, the service must remain *available*, and service changes must often be performed *as soon as possible*.

In recent work [1], [2] we introduced a model-driven perspective for safe and efficient dynamic updates. We proposed a set of criteria of correct dynamic updates with respect to specification changes and elaborated an approach for automatically synthesizing *dynamically updating controllers* from scenario-based assume/guarantee specifications.

Dynamically updating controllers are essentially a structure of two finite state models: the *current controller* and the *updated controller*, implementing the current and updated specification, respectively. Certain states in the current controller, called *updatable states*, are connected to states in the updated controller through *update transitions*. According to this structure, a correct dynamic update occurs when the control of the system execution migrates from the current controller to the updated controller to satisfy the new specification. Such migration occurs when the execution reaches an updatable state, follows an update transition, and continues in the updated controller.

Controllers are executable models that are synthesized from a scanario-based specification. After a change in the specification a new dynamically updating controller is synthesized and deployed for run-time execution.

In this paper we describe an infrastructure supporting this approach, which allows us to deploy and execute newly synthesized dynamically updating controllers on embedded devices. Motivated by scenarios related to space applications, we provide a prototypical implementation in Java for the Lego Mindstorms robot systems. Although we regard the prototype as a work-in-progress, which requires further evaluation, it allows us to present here some of the challenges we encountered, and lesson learned.

The paper is structured as follows. Section II summarizes the foundation of this work with the help of a space robot system as a running example. Section III describes the execution infrastructure and its implementation. Section IV discusses the related work. Section V concludes the paper.

II. Foundations

In this section, we first present a running example of a hypothetical Mars Robot System with its changing specification. We then introduce the foundations of scenario-based specifications and dynamically updating controllers.

A space exploration system used to explore the planet Mars is sketched in Fig. 1. The system consists of a battery-powered exploration robot moving on ground and a charging station. We focus on a simplified scenario that specifies how the robot behaves during the recharging procedure. Using

installed sensors, the robot is able to monitor its own position and recognize where the charging station is located. As it approaches the station, it is able to identify two perimeters of different radii around the charging station: an *outer perimeter*, and an *inner perimeter*. When crossing the outer perimeter, the robot has to reduce its speed. When it reaches the charging station it connects to the power supply. After recharging, it leaves the station and continues its exploration.

Let us assume that a new space mission has been launched and a second robot has landed on the planet. The two robots now share the same charging station and, to avoid collisions, a new coordination mechanism needs to be applied. The new mechanism leads to a change in the specification of the first robot: it introduces a new assumption that between the outer and inner perimeter, the robot now also crosses an *intermediate perimeter*. The change also introduces a new requirement: when entering the intermediate perimeter, the robot must check the status of the charging station. The station may be available for the robot, or it may be unavailable, because it is charging another robot. In the latter case, the robot must wait and after some delay check the station availability again. For simplicity, we will not consider the latter case in the following.

Changes in the specification require the robot system to be dynamically updated, at run-time, in a safe and efficient way.

A. MSD Specifications

In our previous work we show how dynamically updating controllers can be synthesized from specifications given as Modal Sequence Diagrams (MSDs) [1], [2]. MSDs are a formal variant of sequence diagrams introduced by Harel and Maoz [3], similar to Live Sequence Charts (LSCs). An MSD specification in our case consists of a set of universal MSDs, which describe assumptions and requirements in the interactions between the system and the environment. The MSD lifelines represent system or environment components that interact by exchanging messages. Messages have one sender and one receiver; for simplicity, we only consider synchronous messages, where the sending and receiving is a single event, also called *message event*. Messages sent from system components are *controllable events*; messages sent from environment components are *uncontrollable events*.

Messages have a *temperature* and an execution *kind*. The temperature can be either *cold* or *hot* and encodes the safety properties of events in a scenario. Hot messages must not be *violated*, which happens if a message event occurs that is expected only at another point in the scenario. Cold messages can be violated, and this violation terminates the scenario. The execution kind can be either *monitored* or *executed*. Intuitively, if a scenario reaches an executed message, the corresponding message must eventually occur (liveness). If the message is monitored, this message may occur. Hot message are red and cold messages are blue. Monitored messages have a dashed arrow, and executed messages have a solid arrow. For clarity, we also use the labels (h/c, m/e).

As an example, let us consider the *ApproachingChargingStation* MSD in Fig. 1. The MSD contains lifelines that

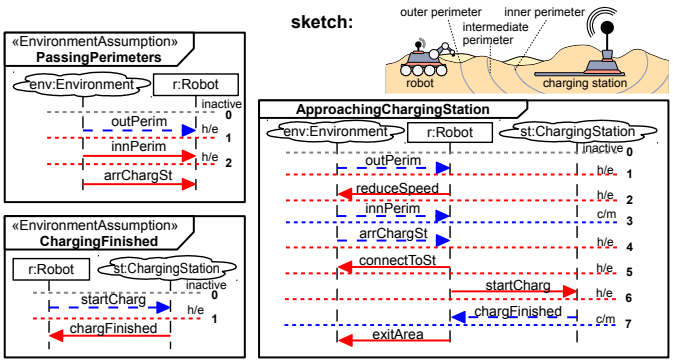

Fig. 1. Current MSD Specification S

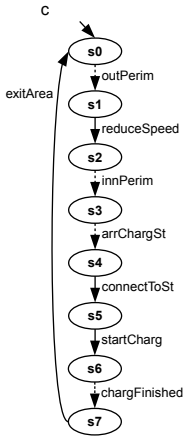

Fig. 2. Controller implementing S

represent the robot, the charging station and the natural environment. The natural environment and the charging station are environment components and the lifelines labels therefore have a cloud-like shape. The first message, representing the *outPerim* message event, is a cold and monitored message, and says that the scenario starts once the robot passes through the outer perimeter. Then the robot must reduce the speed (hot executed message).

We also use MSDs, marked with the *«EnvironmentAssumption»* tag, to explicitly model assumptions about the environment components [4]. The *PassingPerimeters* in Fig. 1 is an example; it specifies that whenever the robot passes through the outer perimeter (*outPerim*), it must eventually also pass through the inner perimeter (*innPerim*), and then reaches the charging station (*arrChargSt*). The assumption MSD *ChargingFinished* similarly says that we assume that after the robot starts the charging process, the charging station will eventually notify the robot that the charging has finished.

We consider the system to be controlled by a *controller*, a finite state machine with two types of transitions: *uncontrollable transitions* and *controllable transitions*, labeled with uncontrollable and controllable events of the specification, respectively. The outgoing transitions of a state are either all controllable all uncontrollable. In states with outgoing

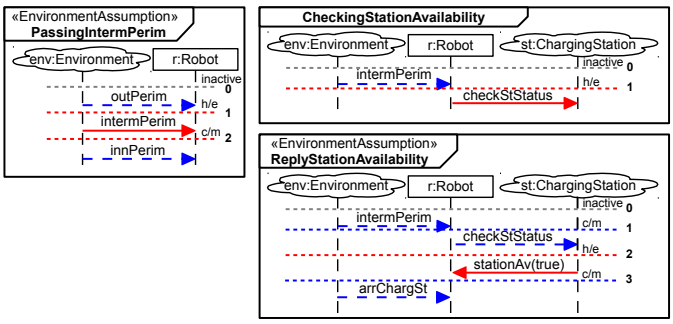

Fig. 3. Additional MSDs that get introduced in Specification S'

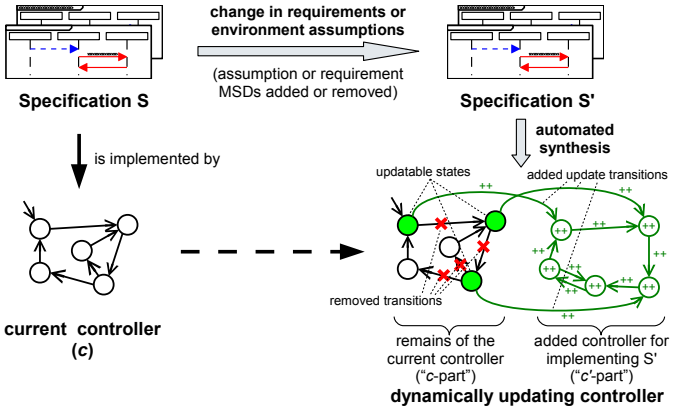

Fig. 4. Synthesis of Dynamically Updating Controllers

Fig. 5. Dynamically Updating Controller of the example

uncontrollable transition the system waits for corresponding environment events; otherwise the system takes a controllable transition and executes the corresponding event. Informally, a controller *implements* a specification if all the resulting sequence of events are valid in that specification.

Figure 2 shows a controller implementing the specification S of the example with the initial state $s0$. The $outPerim$ transition between $s0$ and $s1$ is uncontrollable, meaning that initially the system has to wait for the *outPerim* event to occur. (Other environment events can also occur, but they will not progress the state.) The *reduceSpeed* transition between $s1$ and $s2$ is controllable, meaning that if the system is in state $s1$ it has to execute the *reduceSpeed* event.

As mentioned earlier, the introduction of a second robot leads to changes in the specification. We model such changes by introducing a new MSD specification based on the previous one, in which diagrams can be added or removed.

In our example, the new specification S', is obtained by adding the requirement and assumption MSDs shown in Fig. 3. The MSD *CheckingStationAvailability* specifies the extra check of the availability status of the charging station; the assumption MSD *ReplyStationAvailability* formulates the assumption that the charging station will reply to the robot's status request. (For simplicity, we omit the possibility that it may not be available, along with the behavior required in this case.) Furthermore, we add the environment assumption *PassingIntermPerim* that says that the robot will pass the intermediate perimeter between the outer and inner perimeter.

B. Dynamically Updating Controllers

In our previous work we defined a correctness criterion for dynamic updates based on specification changes [1]. This criterion defines in which state the system is *updatable*, and 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 restarted in its initial state, a dynamic update that follows this criterion is also safe.

We also proposed an approach to automatically synthesize a *dynamically updating controller* from changing MSD specifications. Figure 4 summarizes the approach. It takes as input the current MSD specification S and the new specification S', and the currently executed controller c, which implements S. We then synthesize a maximal controller c' of all valid executions in S'. By comparing the sequence of events generated by c (in the new environment specified by S') with the ones generated by c', it is possible to identify all the executions starting in c that can be completed by c' to satisfy the new specification. We also identify the *updatable states*, states of the current controller in which the system can safely abandon the current obligation and migrate its execution control to the new controller c'. Such migration is obtained by adding *update transitions* from updatable states (in c) to states in c'. For further details refer to our previous work [1].

Figure 5 shows the dynamically updating controller of our example. On the left side of the figure we find the current controller c and on the right side the synthesized controller c'. Three updatable states are identified in this case: $s0$, $s1$, and $s3$. In short, the other states are not updatable, because

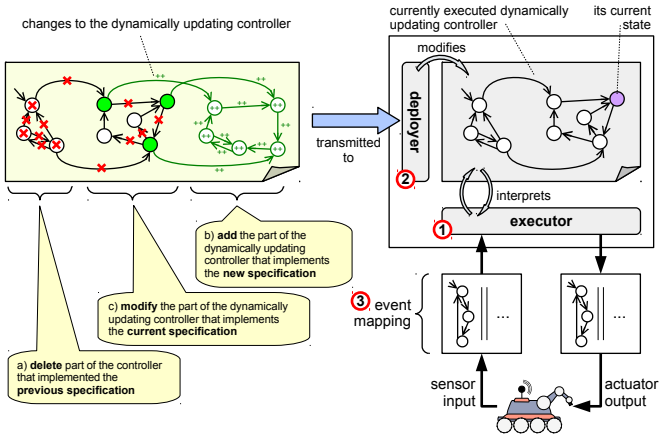

Fig. 6. Overview of the system's execution infrastructure

either something has happened which cannot be completed to a run that will satisfy the new specification, or because it is not know whether something has happened that requires a specific reaction according to the new specification. In state s2 for example, the current controller does not capture whether *intermPerim* has occurred or not; in state s3, however, when *innPermin* occurred, we know that *intermPerim* has occurred.

III. Executing Dynamically Updating Controllers

In this section we describe an execution infrastructure for deploying and executing dynamically updating controllers on a running system. Figure 6 shows an overview. It essentially consists of three parts. The first part **(1)** is the *executor thread*, which executes a current dynamically updating controller. The thread maintains a pointer on the current state of the execution and executes system events or waits for environment events. The second part **(2)** is the *deployer thread*, which is responsible for deploying a new dynamically updating controller, which will be transmitted to the running system.

The third part **(3)** is the *event mapping*, which will translate events from the specification to events of the system. This mapping is necessary, because in the specification, engineers and other stakeholders may formulate the requirements and environment assumptions on a higher level of abstraction; the running system, instead, may have to operate on lower level events. For example, the *outPerim* event in the specification may translate to a specific event of the ultrasonic sensor of the robot; events like *exitArea* for example, may translate to a sequence of signals that the system's microprocessor sends to the motors to rotate and accelerate the robot.

We implemented a prototype of the execution infrastructure for the Lego Mindstorms NXT robot system. The system is based on an Atmel 32bit microcontroller running at 48MHz and addressing 256Kb of flash memory and 64Kb of RAM. Although these strict constraints, the microcontroller is capable of running a small Java virtual machine, called Lejos [1], which will be used in this implementation. The CPU brick can be

[1]http://www.lejos.org

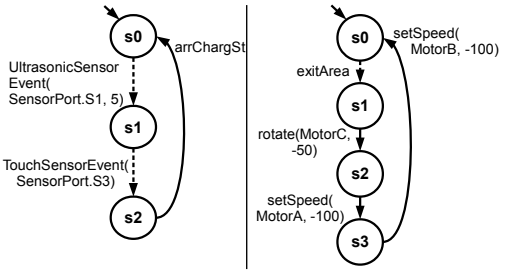

Fig. 7. Using finite state controllers for mapping specification events and concrete events

connected to up to 3 motors, and up to 4 sensors (possible sensors are: buttons, light sensors, sound sensors, ultrasonic distance meters). The system also supports connection via Bluetooth, which will be used to send the dynamic update commands from a remote machine.

The remainder of the section describes the different parts of the execution infrastructure in more detail. We first describe the event mapping, and then the executor and the deployer threads.

A. Mapping specifications events into concrete robot events

As a prerequisite for executing the dynamic controller on the robot system we need to define a mapping from controllable and uncontrollable events in the specification to concrete events occurring in the robot. More precisely, we need to define a way in which the controllable events in the controller can affect the actuators, and how the events monitored by the sensors are mapped to uncontrollable events in the controller.

Also in this case we use finite state controllers to define the sequences of concrete events associated with specification events. There are *sensor input mapping* finite state controllers and *actuator output mapping* finite state controllers, which are all executed in parallel.

Figure 7 shows two examples of such event mapping finite state controllers. On the left, we see the finite state controller that generates the $arrChargSt$ event as the result of receiving a sequence of two lower level sensor events, the first event from the ultrasonic sensor and the second event from the touch sensor. This way, we express that once the distance sensor value goes below a certain threshold, and then a touch sensor event occurs, it means that the robot has arrived the charging station. On the right, we see a finite state controller that generates from the $exitArea$ controllable event a sequence of events which rotate and accelerate the robot.

B. Controller Execution

We now describe how a dynamically updating controller is executed. We define a simple data structure for the controller consisting of a set of states connected with controllable, uncontrollable, or update transitions. Controllable transitions are associated with a specification event that is then translated to events of actuators. Uncontrollable transitions are associated with a specification event that is the result of the translation

from concrete events of sensors. Updatable transitions are ϵ-transitions that are not associated with any event.

The controller is executed by the $executor$ thread following the pseudo-code shown in Algorithm 1.

Algorithm 1 Controller Execution

```
currentState = initialState;
while (true) do
    if isUpdatable(currentState) then
        updatableTransition =
            currentState.getUpdatableTransition();
        currentState = updatableTransition.getTargetState();
    else if hasControllableTransition(currentState) then
        transition = currentState.getContrTransition().any();
        executeSpecificationEvent(transition.getEvent());
        currentState = transition.getTargetState();
    else
        while (true) do
            envEvent = waitForSpecificationEvent();
            uncontrTransitions =
                currentState.getUncontrollableTransitions();
            if uncontrTransitions.contains(envEvent) then
                currentState = uncontrTransitions.
                    get(envEvent).getTargetState();
                break;
            end if
        end while
    end if
end while
```

At the beginning, the current state of the controller is set to the initial state. If the current state is updatable, the corresponding update transition is followed, otherwise the executor checks whether a controllable transition is defined. If this is the case, the specification controllable event associated with such transition is executed (i.e. translated to executable concrete events). If no controllable transitions are defined, then the controller waits for an uncontrollable specification event to occur. Whenever an event is received, the executor checks if, within the uncontrollable transitions of the current state, there is one associated with the received event. If this is the case, this transition is executed.

C. Deploying the dynamically updating controller

We now describe how the dynamically updating controller can be deployed in the system to finally perform the update. First, the dynamically updating controller is parsed to generate a sequence of update commands. An update command can be of the following types: (i) add a state to the current controller; (ii) remove a state from the current controller; (iii) add a transition to the current controller; (iv) remove a transition from the current controller. The update commands identify a state by a unique integer number; a transition is identified by the sending and receiving state as well as the labeling event.

The parsing procedure creates and deletes elements in the order shown in Fig. 6. First **(a)**, it checks the presence of an old controller to decommission and generates the sequence of corresponding remove-transition and remove-state commands. Then **(b)** it generates the sequence of add-state and add-transition commands related to the controller implementing the new specification. Finally **(c)** it generates the sequence of add-transition commands for the update transitions, and remove-transition commands for the outgoing transitions in the current controller that are not used anymore. In all cases, transition removal must occur before state removal, in order to avoid dangling references, and state creation must occur before transition creation. Due to limited computational power of the robot system, breaking down the newly synthesized dynamically updating controller into a sequence of update commands is delegated to a remote machine.

As a second step, the generated sequence of update commands sent from the remote machine is executed in the system. This step is managed by the *deployer* thread. The deployer thread is responsible for performing the corresponding modifications in the controller data structure. Such modifications are performed at run-time, while the system is running, that is, while the executor thread is executing the current controller.

To enable concurrent access to the controller data structure, each state of the controller is associated with a binary semaphore that guarantees mutual exclusion. Whenever entering a state, the executor thread must acquire the semaphore associated with that state before entering it, and it will not release the semaphore until it will exit that state. Whenever the deployer thread is required to perform some operation on an existing state, including adding or removing outgoing transitions, it first needs to acquire the corresponding semaphore before doing any operation on it. This state-level locking mechanism, allows the deployer thread to execute any update command to all states of the controller except on the current state, that is locked by the executor. When an update operation is required on the current state, the deployer thread will wait until the executor thread exits the state, and then perform the required operation.

The Lego NXT brick natively supports communication with a remote machine via USB and Bluetooth. We use the latter to send update command to the robot. To simplify the development of the communication system, all the previously described data structures and update commands are developed supporting their serialization and deserialization over a data stream. In this way, the machine preparing the update can build the structures that are needed, serialize them and send them over the data stream. The deployer thread on the robot can then easily deserialize and perform the updates.

IV. Related Work

The problem of dynamically updating a software system has been addressed in the past and different approaches have been proposed in the area of programming languages [5], [6], [7], [8], and distributed reactive systems [9], [10], [11], [12].

In the area of programming languages, different techniques have been proposed for dynamically updating language-specific constructs such as variables, objects and functions

both for Java applications [5], [6] and for C programs [7], [8]. Our constructive approach to execute dynamically updating controllers, even if implemented on top of a JVM, is general enough to be applied to systems implemented in different programming languages and execution environments.

An important theoretical result is provided in the work by Gupta et al. [13]. They 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. Moreover we provided an automatic technique for identifying such states and to finally execute the dynamic update.

Other approaches proposed different criteria for dynamically reconfigure a distributed systems by replacing at run-time some of their components [9], [10], [11]. These approaches require the system to be in a state where no interactions are currently in place. The same is true for early approaches for dynamically updating only the procedures affected by the changes [14], [15]. Our approach instead allows the update to be performed in a wider set of states (e.g. in all the updatable states) and the performed update will occur more timely.

Research on dynamic software updates is also related to the area of self-adaptive systems. Different approaches have been proposed to modeling, verifying, and execute the dynamic evolution of an adaptive software [16], [17], [18], [19]. Zhang et al. propose a formalism for modeling and verifying adaptive software which requires the manual definition of update points [16]. In our approach, instead, the update points are automatically identified. Adler et al., propose a framework for developing dynamically adaptive embedded systems [17], Bouveret et al., describe a categorical framework to ensure correct software evolutions [18]. Iftikhar and Weyns proposed an interesting approach, called ActiveFORMS, where models of self-adaptive systems are directly executed to perform adaptation [19]. Differently from our approach, in ActiveFORMS dynamic adaptation is obtained by replacing the current model with the new one only when the system is in a quiescence status. This may lead to less timely updates than the one provided by our approach.

V. Conclusion

In this paper we describe a systematic approach for executing dynamically updating controllers that are automatically synthesized from the changes in MSD specification of a system. We implemented a prototype of the approach on top of a Java Virtual Machine for the Lego Mindstorms robot system. This approach, together with the work we presented earlier, is a further step to our vision of having a fully automated framework for designing and executing dynamically updating systems. The engineer has only to focus on providing the MSD specification of the system and of its changes, that will be automatically synthesized in a dynamically updating controllers, deployed, and finally executed.

We applied the approach on a simple robot system. The initial prototype we implemented was useful to have initial insights on the applicability of the approach. Our future work will first focus on further evaluation of the approach on more complex systems. We will also consider on how to synthesize a set of distributed dynamic updating controllers for different system components and how to extend the approach for the design and execution of self-adaptive 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] V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. Brenner, "Formalizing correctness criteria of dynamic updates derived from specification changes," in *Proceeding of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2013)*, 2013.

[3] 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.

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

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] M. U. Iftikhar and D. Weyns, "Activforms: Active formal models for self-adaptation," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS 2014. New York, NY, USA: ACM, 2014, pp. 125–134.