# ActivFORMS: Active Formal Models for Self-Adaptation

M. Usman Iftikhar and Danny Weyns
Linnaeus University, Växjö, Sweden
{usman.iftikhar, danny.weyns}@lnu.se

## ABSTRACT

Self-adaptation enables a software system to deal autonomously with uncertainties, such as dynamic operating conditions that are difficult to predict or changing goals. A common approach to realize self-adaptation is with a MAPE-K feedback loop that consists of four adaptation components: Monitor, Analyze, Plan, and Execute. These components share Knowledge models of the managed system, its goals and environment. To provide guarantees of the adaptation goals, state of the art approaches propose using formal models of the knowledge. However, less attention is given to the formalization of the adaptation components themselves, which is important to provide guarantees of correctness of the adaptation behavior (e.g., does the execute component execute the plan correctly?). We propose Active FORmal Models for Self-adaptation (ActivFORMS) that uses an integrated formal model of the adaptation components and knowledge models. The formal model is directly executed by a virtual machine to realize adaptation, hence active model. The contributions of ActivFORMS are: (1) the approach assures that the adaptation goals that are verified offline are guaranteed at runtime, and (2) it supports dynamic adaptation of the active model to support changing goals. We show how we have applied ActivFORMS for a small-scale robotic system.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*

## General Terms

Theory, design

## Keywords

Self-adaptive systems, formal models at runtime, verification

## 1. INTRODUCTION

Engineering the upcoming generation of software systems, such as networked smart homes and multi-robot systems, and guaranteeing the system goals during operation is complex due to uncertainties resulting from incomplete knowledge at design time. Among the uncertainties are changing availability of resources, dynamic operating conditions that are difficult to predict, and changing goals. Self-adaptation enables a software system to adapt autonomously to deal with such uncertainties. A self-adaptive system typically consists of a managed system and a feedback loop that adapts the managed system according to some goals. In this research, we focus on architecture-based self-adaptation [20, 12, 18] where adaptation is realized with a MAPE-K feedback loop that consists of four adaptation components: *Monitor*, *Analyze*, *Plan*, and *Execute*, complemented with *Knowledge* models of the managed system, its goals and environment [16, 26]. Monitor monitors the managed system and environment through probes, and updates Knowledge models accordingly. Analyze analyzes the data of the knowledge models and checks whether an adaptation is required. If so, it will trigger Plan that will compose a plan with actions that are then executed through effectors by Execute. Central to architecture-based adaptation is the separation of the domain concerns from the adaptation concerns; a recent controlled experiment [25] provides empirical evidence for the engineering benefits for this separation of concerns.

One important challenge in engineering self-adaptive systems is to provide evidence that the system goals are satisfied during operation, regarding the uncertainty of changes that may affect the managed system, its goals or environment [6, 7, 8]. To provide guarantees that the system goals are satisfied, state of the art in architecture-based self-adaptation advocates the use of formal models at runtime as one promising approach. In particular, existing approaches equip the feedback loop with formal models of the managed system and the environment in which it executes. A popular approach is using probabilistic models to model and handle uncertainties. The models are used to verify properties and support decision making about adaptation at runtime.

However, from a study of the state of the art we learned that existing approaches have payed little attention on providing guarantees about the behavior of the adaptation components themselves. For example, important properties of a self-healing system may be: does the analysis component correctly identify errors based on the monitored data, or does the execute component execute the actions to repair the managed system in the correct order? Lack of such guarantees may ruin the adaptation capabilities. In addition, we notice that little attention has been given on support for adaptation for changing goals or adding new goals at runtime. Existing approaches mainly focus on uncertainty with respect to parameters of knowledge models (failure rate of components, availability of a service etc.). Support for adaptation to deal with changing goals or adding new goals typically requires updates of the feedback loop.



*SEAMS'14*, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2864-7/14/06...$15.00
http://dx.doi.org/10.1145/2593929.2593944

In this paper, we propose Active FORmal Models for Self-adaptation (ActivFORMS). ActivFORMS contributes to the state of the art with an approach that guarantees the verified adaptation behavior at design time and provides first-class support for dealing with changing goals at runtime. The approach uses an integrated formal model of the complete MAPE-K loop, i.e., models of the knowledge and the adaptation components. The integrated formal model is directly executed by a virtual machine at runtime. We refer to this integrated formal model as *active model*. The active model can be dynamically changed with changing goals. The contributions of ActivFORMS are twofold. First, ActivFORMS assures that the goals that are verified offline are guaranteed at runtime. Second, the approach supports dynamic changes of the active model to support changing goals. This contrasts to existing approaches that provide guarantees during design, but require additional efforts to transfer the design into an actual implementation and maintain guarantees. In this paper, we focus on adaptation with MAPE-K feedback loops, but ActivFORMS can be applied to other types of feedback loops that can be modeled using the formal approach.

We evaluated ActivFORMS for a small scale system in which robots perform transportation tasks in a warehouse environment. The adaptation goal is to enable robots to adapt their behavior when a lane in the warehouse is temporally closed, for example for maintenance. We show that equipping each robot with an active model guarantees that the robots adapt their behavior correctly. The approach dynamically adapts the adaptation components to deal with potential deadlock due to a changing warehouse layout.

The remainder of this paper is structured as follows. Sect. 2 discusses the state of the art on the use of formal models for self-adaptation at runtime. In Sect. 3, we briefly introduce the robotic system. Sect. 4 presents the ActivFORMS approach and explains how we applied the approach to the robotic system. We reflect on the tradeoffs and restrictions of the approach in Sect. 5. Finally, we draw conclusions and outline possible future research in Section 6.

## 2. STATE OF THE ART

A recent systematic literature survey [24] covering the main software engineering venues between 2000 and 2012 identified a total of 75 papers that use formal methods in architecture-based self-adaptive systems. Among the primary studies, 25 study formal methods at runtime. We discuss a representative set of these papers and focus in addition on recent research results. We conclude with pointing out a number of interesting challenges in this area.

Back in 2002, Garlan and Schmerl [13] proposed an approach for model-based runtime adaptation of self-healing systems. Architecture models specified in Acme are checked via Armani, which evaluates first order constraints on the fly as properties of the architecture change. When problems are detected Armani triggers a repair engine to look for a repair strategy. This work laid the basis for the Rainbow framework [12].

[27] presents a process to create formal models for adaptive systems, verify the models and automatically translate the models into executable programs. The authors use Petri Nets and linear temporal logic to provide assurances for the system goals, and model-based testing to guarantee conformance between the models and programs. In follow up work [28], the authors model a dynamically adaptive program as a collection of (non-adaptive) steady-state programs and a set of adaptations that realize transitions among steady state programs in response to environmental changes. To handle the state explosion, the authors propose a modular model checking approach. Linear Temporal Logic (LTL) to specify properties of the non-adaptive portions of the system is combined with A-LTL (an adapt-operator extension to LTL) to concisely specify properties that hold during the adaptation process. This work provides an advanced approach for modular verification, however, its application at runtime needs further study.

[9] uses a probabilistic model (discrete time Markov chain) to represent an abstraction of the possible execution flows of a system at runtime. The probabilities that represent uncertainties are dynamically updated based on observations, using a Bayesian estimator. The probabilistic model can be used by a feedback loop to detect requirements violations and optimize the realization of system goals dynamically.

[11] proposes an approach for efficient runtime verification of reliability requirements. The proposed solution considers two distinct steps: pre-computation at design time and verification at runtime. The output of the pre-computation step is a set of symbolic expressions, which represent satisfaction of the requirements. The verification step then evaluates the formula by replacing the variables with the runtime values gathered by monitoring the system.

[10] presents a quantitative approach for making adaptation decisions under uncertainty, called POISED. POISED builds on possibility theory (that is grounded in fuzzy mathematics) to assess both the positive and negative consequences of uncertainty. POISED makes adaptation decisions (runtime reconfiguration of its customizable software components) that result in the best range of potential behavior, improving a software system's quality of service.

[5] presents an advanced approach for self-adaptation to achieve QoS for service-based systems. Formally specified requirements are automatically analyzed to identify and enforce optimal system configurations by adapting service selection and resource allocation. The approach realizes a feedback loop based on MAPE-K, see Fig. 1. The Knowledge part of the feedback loop is modeled

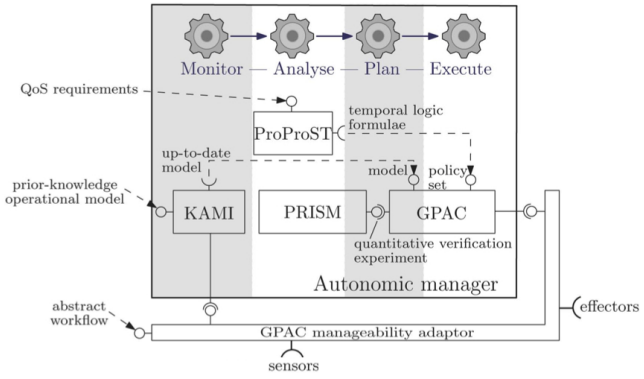


**Figure 1: MAPE-K realization in [5]**

using different formal languages. The models are used by different tools to assure optimal reliability and performance requirements. As shown in the figure, the MAPE loop is realized by a series of tools that are glued together.

[14] introduces adaptive model-driven execution to mitigate uncertainties. In this approach, a Markov decision model of the system is generated from UML interaction diagrams. The model specifies the probability distribution of the different execution paths of the system. The model is executed by an interpreter that drives the execution of the system to guarantee the highest utility for a set of quality properties. The approach uses an embedded model to realize adaptation, but the concerns of the domain and adaptation are not clearly separated. The adaptation logic, which is encoded in the interpreter projects the possible future paths in the model to select the next action on the path with the highest utility.

Two interesting recently proposed approaches are [19] that focus

on dynamic updates to deal with changing assumptions and requirements at runtime in time-critical systems and [23] that proposes the EUREMA approach that realizes self-adaptation based on so called executable runtime mega-models.

**Summary.** To provide guarantees in self-adaptive systems, the use of formal models has gained increasing attention. Approaches that provide assurances by construction underpin the importance of formal methods. However, these approaches require additional efforts to provide guarantees of the actual implementation. Such efforts may be substantial to handle changing goals at runtime. For approaches that employ formal models at runtime, quantitative approaches are dominant and a number of studies support runtime verification. Virtually all studies focus on modeling the manged system, its environment, and the system goals (or parts of these). These models are then used by adaptation logic (i.e., the adaptation components) to reason about the system behavior and support analysis and decision making of adaptation actions. However, formal modeling of the adaptation components themselves and guaranteeing the required properties of the adaptation behavior has gained little attention. In a number of approaches, the realization of the adaptation logic includes tools, a prominent example is shown in Fig. 1. However, the behavior of the MAPE components and the integration of tools is often not formalized or verified. Furthermore, more research is required on adaptation to handle changing adaptation goals or adding new goals to the system at runtime, which represents an important class of uncertainty. Handling such changes typically requires dynamic updates of the adaptation components (and probably the managed system), which may require human involvement. Finally, model checking techniques are known to be computationally expensive, as they suffer from the state space explosion problem. Any solution based on such techniques, either used at design time or runtime is restricted with respect to providing guarantees in terms of system size. This also applies to ActivFORMS. To that end, research is required to realize scalable runtime verification for self-adaptive systems. However, this challenge is not the focus of the research presented in this paper.

# 3. ROBOTIC SYSTEM

Before we present ActivFORMS, we first briefly introduce the case study that we used in this research. The application consists of a set of robots that have to perform transportation tasks in a warehouse environment. Fig. 2 (left hand side) shows the user interface of the robotic system in simulation.

The map layout consists of a graph of connected nodes. In the example there are three source (or pick) locations (on the right hand side of the map marked with S) and two drop locations (on the left hand side marked with D). There are also two park locations (the zones at the top and bottom of the layout, see the figure). The tasks in this setting are performed by two robots. The robots receive tasks from a task managing system, or tasks can be manually added to the system. A task consists of picking a load at a pick location, drive to the drop location, and drop the load there. To avoid collisions and deadlock when they perform tasks, the robots can synchronize with one another to lock the next node they plan to visit. An idle robot can park at one of the park locations.

The robot system can operate in two modes: standard mode and shipping mode. In standard mode, the robots perform regular transportation tasks within the warehouse. In shipping mode, the robots have to perform tasks to load or unload a truck that can park at the drop location at the top. Mode changes are typically planned in advance, but an operator can also switch modes manually. In the depicted situation, the system is in standard mode (indicated by the light gray shade of the lane connected to the park location for trucks that is currently not accessible for the robots).

In this paper, we use self-adaptation to deal with lanes that have to be closed temporally in the warehouse, e.g., to perform maintenance tasks or to solve a problem with a robot. A lane can only be closed if none of the robots is depending on the lane for their current tasks. Closing a lane may also create the risk for deadlock, which needs to be anticipated.

To facilitate self-adaption capabilities, the robot program offers a monitor API that allows to retrieve the status of the robot (current position, current task, locked node, etc.) and an effector API that allows to perform adaptions of the robot (disable and enable a lane in the map of the robot, add and remove an element on the map, lock a node, etc.).

We are currently testing the scenarios with Turtlebot 2 robots (http://www.willowgarage.com/), see Fig. 2 (right hand side). The robot program to perform the transportation tasks is written in Java and is deployed on each robot. This program interacts with a local Python script that sends basic movement commands to the robot hardware using the Robot Operating System (ROS) API.

# 4. APPROACH

We now present the ActivFORMS approach. Fig. 3 shows the primary modules of ActivFORMS. The approach is in line with the three layered reference model for self-adaptive system proposed by Kramer and Magee [18]. The managed system realizes the domain functionality for users. In this research, we assume that the managed system is prepared to enable monitoring of relevant state and executing adaptation actions. Preparing the managed system for instrumentation to monitor and adapt the system is a research subject in its own right and out of scope of this paper. In the robotic system, the managed system is the robot program that enables the robot to perform transportation tasks. As explained in the previous section, the robot program provides monitoring and effector API's to facilitate extensions with self-adaptation capabilities. We now present the two central components of ActivFORMS: the active model engine and goal management.

## 4.1 Active Model Engine

The active model engine consists of two parts: an integrated formal model that realize a MAPE-K feedback loop, i.e., the active model, and a virtual machine that can execute the active model.

### *4.1.1 Active Model*

In this research, we model feedback loops using networks of timed automata [2]. A timed automaton is a finite-state machine that models a behavior, extended with clock variables, which are used to synchronize behaviors. Automata can communicate through channels. There are two type of channels, binary channels and broadcast channels. For a binary channel, a sender x! can synchronize with a receiver x? through a signal. If there are multiple receivers x? then a single receiver will be chosen non-deterministic. The sender x! will be blocked if there is no receiver. A broadcast channel sends a signal to all the receivers, and if there is no receiver, the sender will not be blocked. Behavior specifications can be complemented with expressions specified in a C-like language to define data structures (struct concept) and functions. Goals can be expressed in timed computation tree logic expressions (TCTL). TCTL expressions describe state and path formulae that can be verified, such as reachability (a system should/can/cannot/... reach a particular state or set of states), liveness (something eventually will hold), etc. We use Uppaal [1], a model checking tool that supports modeling of behaviors and verification of properties.

Fig. 3 shows an overview of the structure of the active model

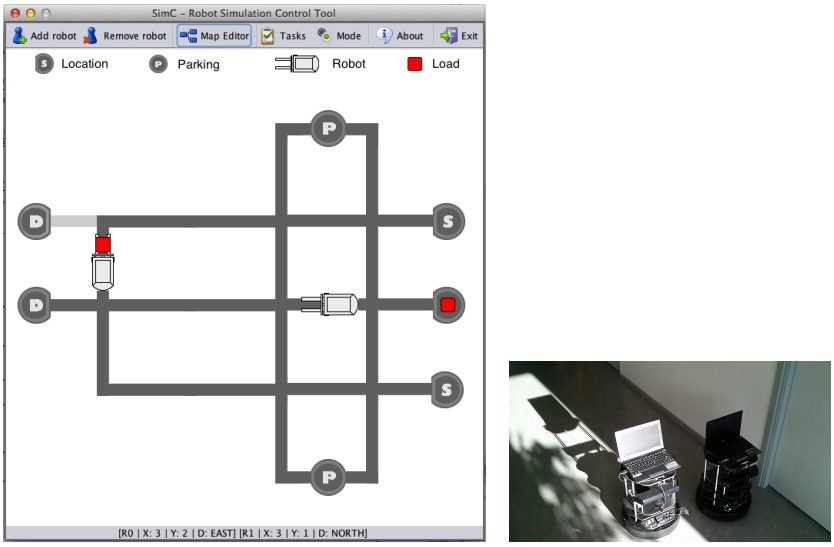


**Figure 2: Left: User interface to the robot simulator. Right: Two Turtlebots in action.**

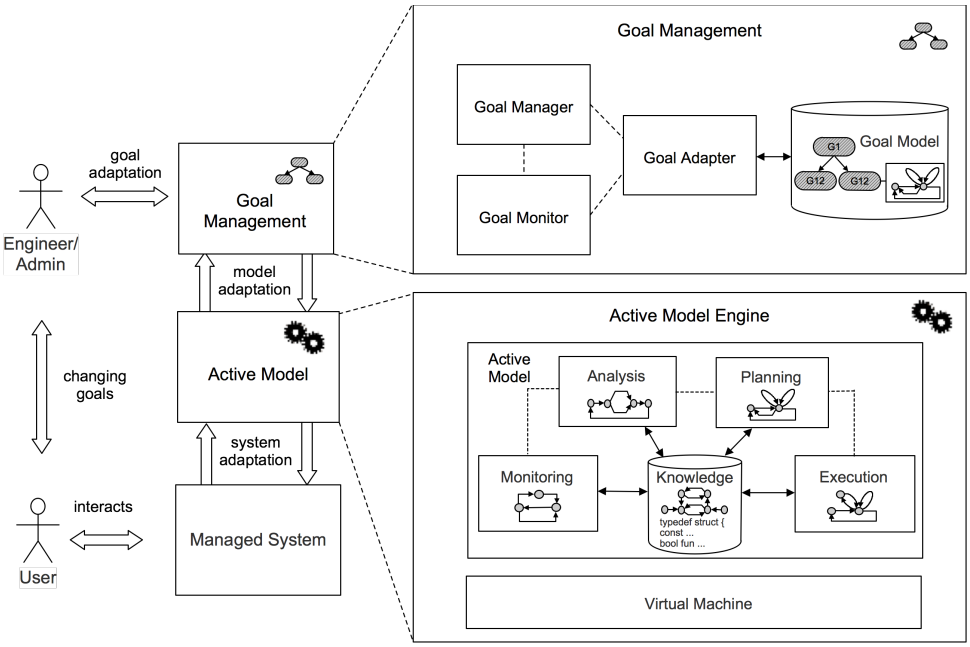


**Figure 3: ActivFORMS approach**

of a feedback loop design. The model consists of a network of timed automata (also called behaviors). The monitor behavior can receive data from probes that connect with the managed system. The execution behavior can send adaptation actions to effectors. The monitor, analyze, plan, and execute automata can interact directly via channels, or indirectly via reading and writing data in the knowledge. Knowledge can be represented with automata or with data structures (struct), or a combination of both. Fig. 4 shows a concrete example of an automaton of a robot that deals with the planning to disable a lane in the warehouse.

Planning starts when the automaton receives a signal from analysis (*planningDiableLane[RiD]?*). Planning first initiates a plan (*initiatePlan()*) and then checks whether the robot depends on the lane that has to be disabled, i.e, it may currently travel on the lane or it may need the lane to perform its current task (condition *dependOnLane()*). If that is the case, planning adds a step to the plan to let the robot wait until the condition no longer holds (*addPlanStepWait()*). Planning then moves on to *PlanDisable*. If the robot does not depend on the lane, planning immediately moves to *PlanDisable*. Planning then adds a step to the plan to disable the lane (*addPlanStepDisable()*), which completes planning. Finally, planning signals the execute behavior to execute the plan (*execute[Rid]!*).

**Formal Guarantees.** During design of the formal model, the required adaptation goals can be verified. To that end, the formal model of the MAPE-K feedback loop has to be connected with a models of the managed system. Evidently, the guarantees obtained

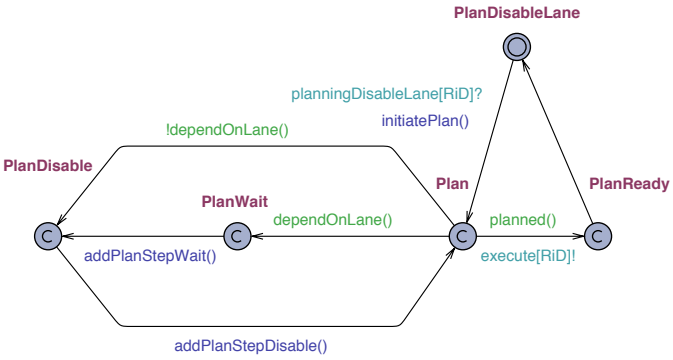


**Figure 4: Plan automaton of a robot**

from verification only hold to the extent that the implementation of the managed system conforms to the model of the system that is used for verification. Such conformance can be tested, for example with model-based testing techniques [21]. We illustrate verification with an example for the robotic system.

```
Monitoring(1).RequestToDisableLane
 && knowledge[1].disabledLane == Lane_cd
  --> Execution(1).DisableLane
  && knowledge[1].disabledLane == Lane_cd
```

The goal allows verifying that when the monitor behavior of robot with ID 1 receives a request for disabling a particular lane, the execution behavior will eventually adapt the managed system accordingly.

### *4.1.2 Virtual Machine*

In ActivFORMS, the formally verified model can directly be executed to realize self-adaptation using a virtual machine. The virtual machine can perform the following functions: initiate model, execute model, interact with the managed system and the environment, verify goals at runtime, and update running models when requested. We discuss these functions in detail.

**Initiate Model.** When the virtual machine starts[1], it first translates the active model (network of automata) to an internal graph representation. Concretely, each node of an automaton becomes a node of a graph for that automaton; links between the nodes become edges between the corresponding nodes. Operations such as checking guards, updating state, etc. are translated into task graphs that are associated with the corresponding nodes and edges. Communication between automata (signals) are integrated in the task graphs of the edges or nodes that send and receive signals.

Fig. 5 shows an excerpt of the internal representation of the formal model of a robot. Fig. 5(a) shows the analysis automaton of a robot, (b) shows an excerpt of abstract syntax tree of the transition between the nodes *Analyzing* and *DisableLaneRequest*, and (c) shows the task graph generated for the guard *DiableLane == matchRequest(request)*. The task graph shows the subsequent atomic tasks that need to be executed to check the guard.

When the active model is translated to the internal representation, the state of each graph and the global state is initiated and the model is then ready for execution.

**Execute the Model.** Model execution conforms to the semantics of networked timed automata. The execution of the active model is

[1]The virtual machine is implemented in Java and can be started with the ActivFORMSEngine class.

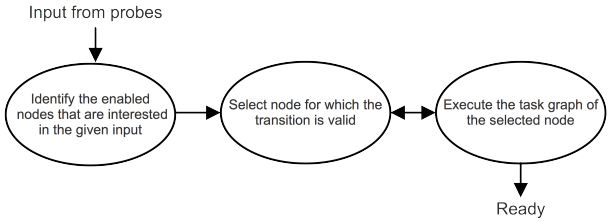


**Figure 6: Input triggered execution**

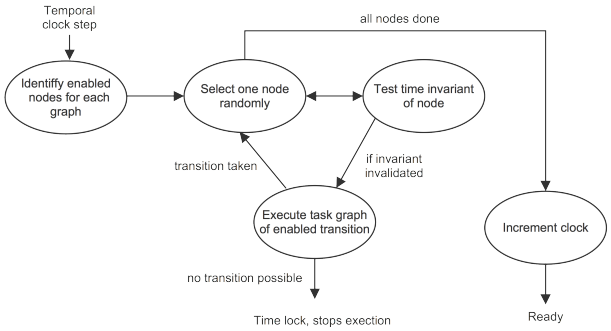


**Figure 7: Time triggered execution**

triggered either by input from the managed system or the environment,[2] or by time.

Fig. 6 shows the execution steps of an input triggered execution. When the virtual machine receives input, it first identifies the enabled nodes that are interested for the given input, which will be one or more nodes of the monitor behavior of the feedback loop. Next the virtual machine selects a node for execution.[3] If the transition is valid, the task graph of the selected node is executed, otherwise another enabled node is selected for execution. The execution of the task graph may trigger a subsequent behavior, e.g., a monitor may trigger an analysis and so forth. If there exists no valid node, no transition will be taken. This may point to an inconsistency between the model of the managed system that was used during the design of the managing system and the implementation of the managed system. An example of an input triggered execution in the robotic system is a monitor behavior that receives a signal from a probe to disable a lane and starts processing this request.

Fig. 7 shows the execution steps of a time triggered execution. The virtual machine maintains an internal clock that increments with time steps. The real time that corresponds with each time step can be configured in the virtual machine engine. In the following example, each step of the clock corresponds with 100 ms.

```
engine.setRealTimeUnit(100);
```

In line with the semantics of timed automata, for each time step, the virtual machine identifies the enabled node for each automaton and checks whether the time step would invalidate the time invariants of the enabled nodes. The virtual machine will then execute

[2]The active model interacts with the managed system and the environment via signals that communicate with probes and effectors. We explain the details of interaction via probes and effectors below.

[3]We limit the explanation to input triggered execution for binary channel semantics, that is, a signal that is sent from a probe synchronizes non-deterministic with one enabled location of a monitor automaton. The virtual machine also supports broadcast channels, where a signal can trigger multiple enabled locations.

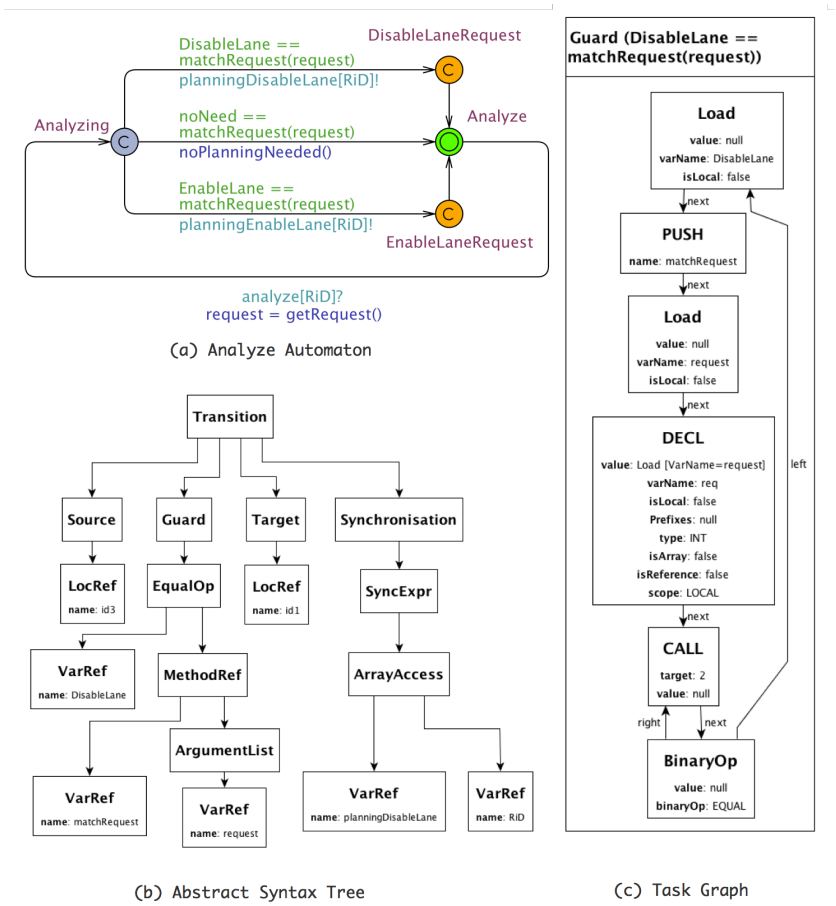


**Figure 5: Excerpt of internal model representation**

the task graphs of these invalidated nodes in non-deterministic order. If any of the nodes with a time invariant is not able to make a transition (due to a design flaw), time can no longer progress, which causes a timelock. In that case, a time lock exception will be thrown and execution of the model terminates. If no invalidated nodes exist, the clock will be increased and the execution step ends. An example of a time triggered execution in the robotic case is an execute behavior that executes the first step of the plan to disable a lane (the robot depends on the lane, see Fig. 4 and Fig. 11) and periodically checks whether the robot still depends on the lane.

**Interaction with Managed System and Environment.** The virtual machine interacts with the managed system and the environment through probes and effectors. For example, Fig. 8 shows two sequence diagrams that illustrate the interaction between the effector of a robot and the ActivFORMS engine to enable lanes.

To communicate with the virtual machine, the effector has to implement the *Synchronizer* interface and register to the virtual machine for channels of enabling lanes (Fig. 8(a)). When the execute behavior of the virtual machine wants to disable or enable a lane, it will synchronize with the effector using *readyToReceive()* (Fig. 8(b)). The execute behavior will then send the data of the adaptation action to the effector using *receive()*. Once the effector knows which lane' status has to be changed, it will execute the adaptation action using *performAdaptation()*. Probes work in a similar manner. The following excerpt shows how a probe communicates the updated position of a robot to the monitor.

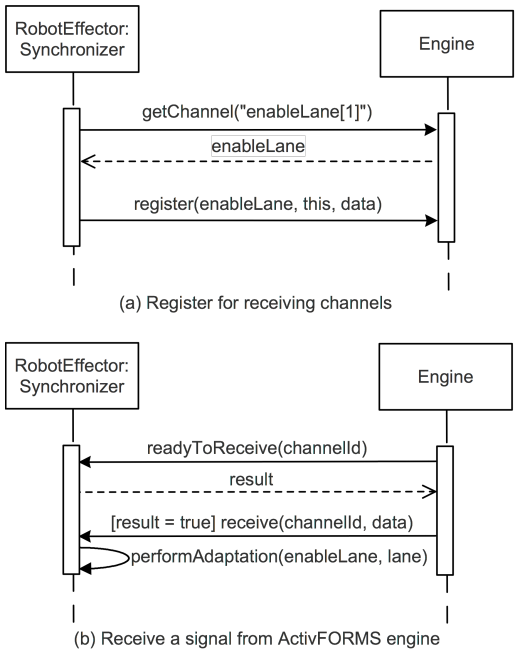


**Figure 8: Interaction between an effector and the engine**

```
engine.send(updatePosition, synch,
            "position.x=" + position.x,
            "position.y=" + position.y);
```

The parameter *updatePosition* represents the channel id, *synch* is a reference to return an acknowledgment when the *updatePosition* message is accepted. The last two parameters contain the data that the probe wants to send to the managing system, i.e., the current robot position. The virtual machine will process this message and execute the task graph of the enabled node of the monitor.

**Verify Goals at Runtime.** As formal verification is costly and exhaustive verification is hard to achieve, in particular when conditions are uncertain, verification at runtime may increase the evidence for assurances. Runtime verification can exploit the concrete information of the system and its environment that is available. ActivFORMS can exploit the direct availability of the formal model at runtime to support runtime verification. However, efficient runtime verification for self-adaptation remains a complex problem with many challenges. In the current version of ActivFORMS, we take a first step towards support for runtime verification. Concretely, the ActivFORMS engine currently supports the verification goals that can be specified as boolean expressions. To that end, the user has to specify the goal and load it into the virtual machine. The virtual machine will verify the goal in each execution step and notify the user whether the goal is satisfied or violated.

For example, a requirement for the robotic system is that a robot should never drive on a lane that is disabled:

```
notOn_disabledLane =
 "DISABLED_LANE == true &&
   knowledge[1].currentLane !=
      knowledge[1].disabledLane";
```

The virtual machine offers a method *addGoal()* to register goals:

```
engine.addGoal(notOn_disabledLane, client);
```

The *client* is the component that has an interest in the state of the goal.[4] When a goal is registered the virtual machine convert it into task graphs. After each transition, the virtual machine executes the task graphs of all registered goals and notifies the user whether the goals are satisfied or violated.

**Changing Active Model at Runtime.** There are two important motivations to provide support for changing the active model of the feedback loop at runtime. First, it enables efficient verification at runtime. Using specific models for the adaptation components that are tailored for different goals keeps the models small, which supports efficient verification. In the next section, we show how the goal manager automatically changes models of the feedback loop to deal with changing goals. A second important motivation is to support deployment of new models at runtime to deal with new goals. This latter typically involves humans in the loop. While this feature is supported by ActivFORMS, it is not the main focus of the research presented in this paper.

Fig. 9 shows the subsequent steps to change (parts of) a running active model. To start the change of an active model, the different parts of the model that need to be changed are loaded into the virtual machine using the method:

```
engine.changeModel(model);
```

The virtual machine then translates the model to the internal graph representation. The remaining steps to complete the dynamic

[4]In section 4.2, we will see how the Goal Manager of ActivFORMS serves as client to support changing goals.

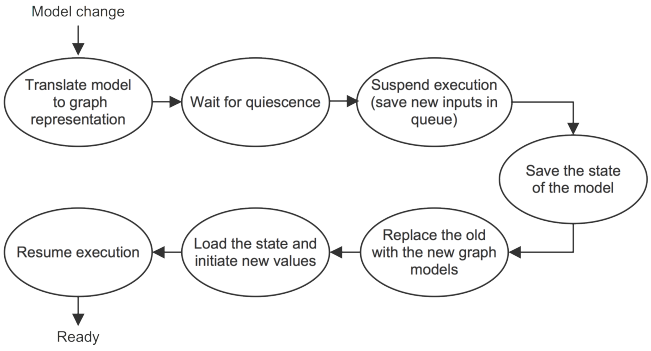


**Figure 9: Change of the formal model at runtime**

change of the active model essentially follow the classical process of runtime updates based on quiescence [17]. First, the virtual machine waits until the model reaches a quiescence state (i.e., no input or time triggered execution is ongoing). The virtual machine then suspends the execution. The state of the model is saved and new inputs are stored in a buffer. Subsequently, the old model is replaced by the new model. The saved state is restored and new variables are initiated. Finally, execution is resumed.

## 4.2 Goal Management

Goal management deals with adaption issues that cannot be handled by the current active model. Goal management consists of four key parts (see Fig. 3): goal model, goal monitor, goal adapter, and goal manager. We discuss each of them in detail.

**Goal Model.** The goal model represents the adaptation goals. We use tree-based models to specify goals. The goals at the bottom level of each subtree have associated models to realize adaptations. Fig. 10 shows an excerpt of a goal model for a robot.

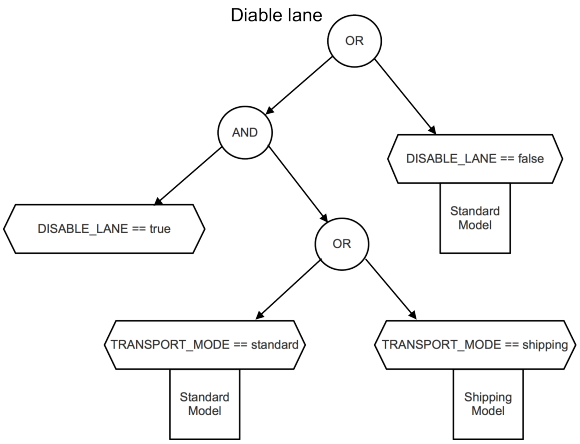


**Figure 10: Goal model example**

The figure shows a goal tree to support adaptation for disabling lanes. The subtree on the left hand side combines *DISABLE_LANE == true* with two *TRANSPORT_MODE*, *standard* and *shipping* respectively. With each mode a corresponding formal model is associated. To support disabling of lanes in the standard mode, the virtual machine needs to execute the *Standard model*. However, in shipping mode, the *Shipping model* needs to be executed. To illustrate the need for different models, consider Fig. 11 that shows the models for the execute behavior for the two modes.

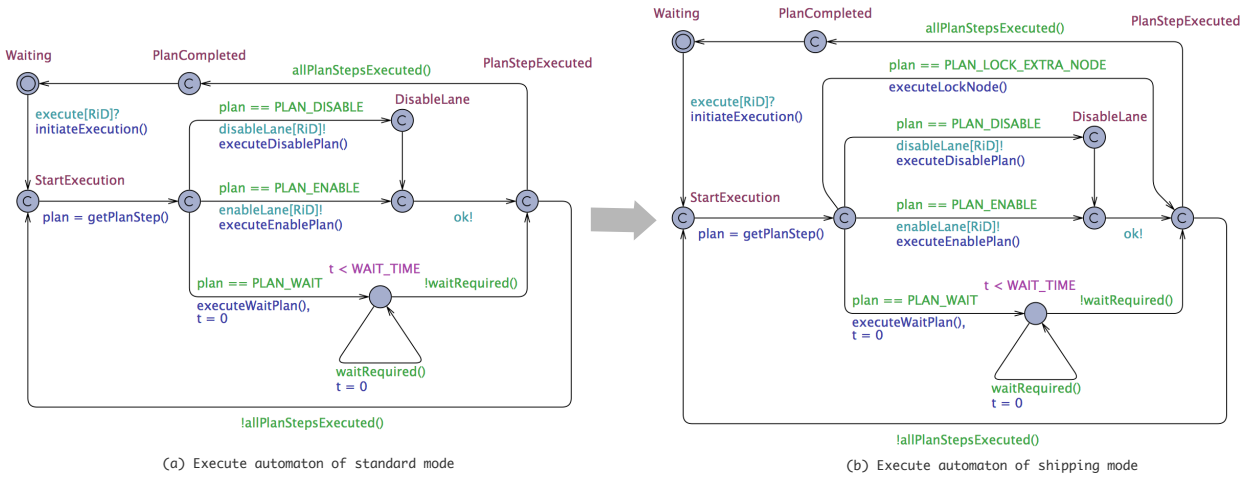


**Figure 11: Execute behaviors for a robot in two operation modes**

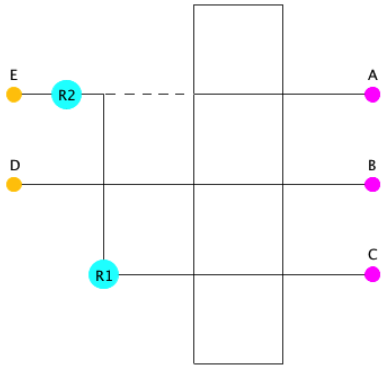


**Figure 12: Deadlock scenario in robot system**

For standard mode, the execute behavior executes the steps of the plan to disable a lane (see the plan behavior of Fig. 4). However, in shipping mode, the map layout changes (a new lane and drop location is added, see Fig. 2), which creates new types of deadlock when a lane would be disabled.

Fig. 12 shows a schematic scenario for the robots in shipping mode. Assume the robots would have disabled the lane marked in dotted line. When robot R1 wants to drop a load at E and both robots have locked one node ahead, they end up in deadlock. To anticipate this problem, the new formal model of the feedback loop is required before the lane is disabled that avoids such deadlock situations. A concrete solution is to ensure that each robot locks two nodes before a lane can be closed. Therefore, the plan and execute behaviors have to be updated. Concretely, the planner in shipping mode has to add a extra step to the plan to lock an extra node. Fig. 11 shows the extra step that the execute behavior executes to lock an extra node, i.e., *executeLockNode()*. Locking an extra node constraints the mobility of the robots (it constraints path selection to drive), and should therefore only be applied when there is a request for disabling a lane in shipping mode.

**Goal Monitor.** The goal monitor monitors the status of the goals. To that end, the goal monitor adds the goals to the virtual machine and registers as client for notification of the status of the goals. The virtual machine keeps the goal monitor updated about the status of the goals. The goal monitor in turn will inform the goal adapter of any goal state changes and keep the goal manager up to date about the status of the goals (goal manager is discussed below).

**Goal Adapter.** The goal adapter is the heart of goal management. When the goal adapter is signaled by the goal monitor about a change of goals, it consults the goal model and search for a matching model that satisfies the changing situation. If the model associated with the changing goal differs from the currently deployed model, the goal adapter starts updating the current model with the new model at the virtual machine. If the model does not differ no further action is required. As an example, consider the Disable Lane goal shown in Fig. 10. If the robot system is running in standard mode and a lane is disabled, the standard model is running (left subtree of the goal). However, when the system switches to shipping mode, the running model has to be changed to shipping model. Once the new model is deployed, the system is ready to deal with deadlock when disabling lanes in shipping mode.

When the goal adapter finds no matching model that satisfies the changing goals, it will notify the goal manager. In this case, the adaptation goals cannot be satisfied and the goal manager will inform the system administrator.

**Goal Manager.** The goal manager offers support for three primary functions: inspecting the active model and its ongoing execution, monitoring and updating goals, and updating the goal model. In our current implementation, the ActivFORMS User Interface connects with the goal managers of the different nodes of the system. The user interface enables the system admin (or engineer) to perform the functions of the goal managers remotely. Fig. 13 shows the ActivFORMS User Interface.

The user interface allows a system admin to connect with goal managers of different nodes using the *Connect* button. In the snapshot, the user interface is connected with two robots. The main pane shows basic info about the robots and the status of their goals.

Clicking the *glasses* symbol for a robot shows the running model of the feedback loop of that robot, as illustrated in Fig. 14. The pane on the right hand side shows the current state of all data variables. The pane on the right hand side shows the models in action.

Fig. 15 shows the window that opens when the *Update Goals* button of the user interface is selected. The system admin can select a goal for any of the connected nodes in the left pane and edit

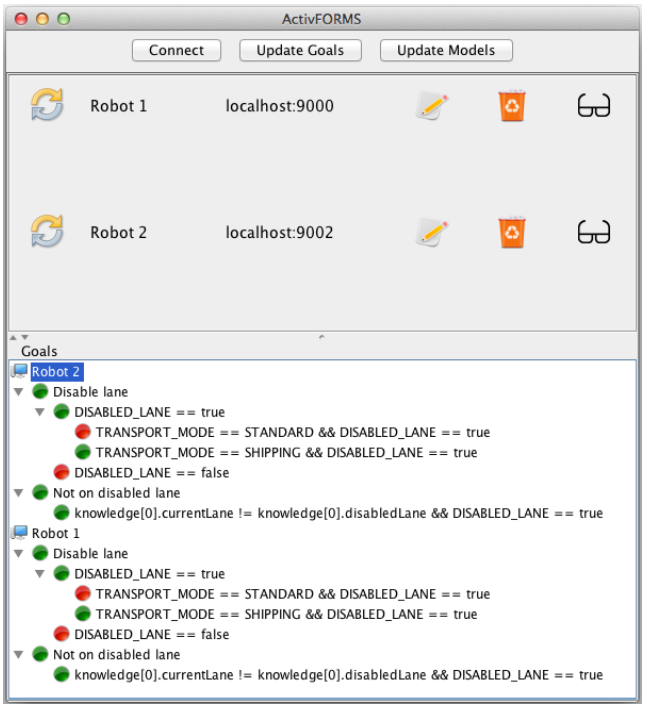


**Figure 13: ActivFORMS User Interface**

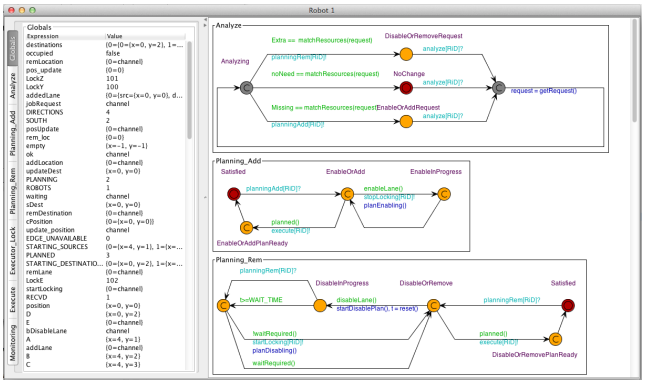

**Figure 14: Illustration of an executing active model of a robot (red locations are current active locations)**

the goal in the right pane. A new model can be selected and associated with the selected goal (*Save* button). Furthermore, new goals can be added or goals can be removed. All goal changes are directly forwarded to the corresponding goal manager to load the goal model. The model will be executed once it is selected by the goal adapter. To manually update the running model, the admin can select the *Updates Models* button in the user interface. We do not further discuss this feature here.

## 5. DISCUSSION

By directly executing the active model at runtime, ActivFORMS provides guarantees of the self-adaptive behavior and supports dynamic updates of the feedback loop. Active models go beyond the notion of model@runtime, which is defined [3] as a causally connected self-representation of the associated system [...] from a problem-space perspective. An active model is the *engine* that ex-

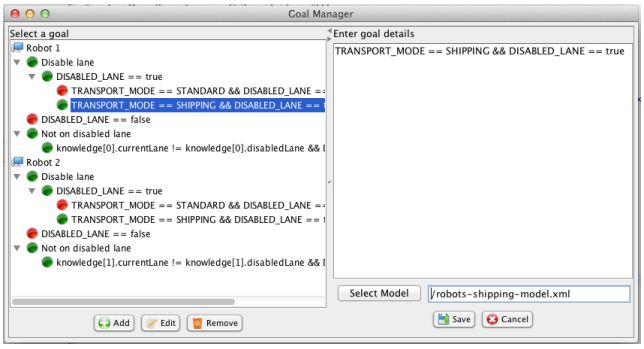


**Figure 15: User interface for adding new goals**

ecutes self-adaptation using a self-representation. Formalizing the distinct behaviors of a feedback loop supports fine grained verification of the correctness of the adaptation behaviors. Furthermore, modeling goals as first class entities and handling changing goals by dynamically changing the formal models of the adaptation behaviors supports small models that can be verified efficiently (at design time and potentially at runtime).

However, the proposed approach has a number of tradeoffs and restrictions. First, the approach requires expert knowledge to design and change the formal models, which is a characteristic of every approach where designers have to use formal methods. ActivFORMS uses timed automata and TCTL, which offer an accessible notation. To further support designers with modeling MAPE-K feedback loops, we have developed a set of templates for designing the behaviors of the MAPE-K components.[5] These templates define abstract automata that can be instantiated and extended for the domain at hand. The templates were derived from experience with modeling MAPE-K feedback loops for different applications, e.g., [22, 15]. Nevertheless, formal modeling remains a tedious task. In our future work, we plan to explore how the underlying formalism could be hided from designers. An inspiring approach is proposed in [14]. Second, in this paper, we have used ActivFORMS to realize MAPE-K loops, but ActivFORMS can execute other types of feedback loops [4] as long as the components of the loops can be modeled correctly with timed automata. Third, the approach relies on models. However, accurate information to the design the models may not be available at design time. To support modeling of uncertainty of the environment and the system in the knowledge part of the feedback loop, we are currently working on extending ActivFORMS with probabilistic timed automata. Fourth, timed automata may not be an appropriate language for modeling the behavior of the feedback loop for particular types of systems; an example is a system that requires continuous adaptation features. Fifth, ActivFORMS approach is not tested yet for medium or large-scale systems. One feature of the approach to handle scalability is dynamic switching of models to handle different goals, which keeps the running models small. Nevertheless, further study is required to study the scalability of the approach. Sixth, ActivFORMS introduces some overhead. The memory required to launch the virtual machine engine and load a active model is approximately 100 MB, depending on the size of the active model that is used. Performance overhead may be an issue, as the virtual machine has to check the validity of the transitions of the enabled nodes in each execution step. For the robotic system, the performance overhead was minimal, but this might be different for other domains.

[5]Available at the ActivFORMS website:
http://homepage.lnu.se/staff/daweaa/ActivFORMS.htm

## 6. CONCLUSIONS & FUTURE WORK

In this paper, we presented ActivFORMS, a formal approach for self-adaptation. ActivFORMS distinguishes itself for existing approaches in two ways. First, the formally verified model of the complete feedback loop is directly executed, which guarantees the verified adaptation goals at runtime. This contrasts in particular to existing approaches that provide guarantees during design, but require additional efforts to transfer the design into an actual implementation and assure guarantees. As the active model is directly executed in ActivFORMS, the approach does not require coding. We recently performed a series of case studies in the context of a Master course on Adaptive Systems. Initial results show that the total time for adding a self-adaptation property to a legacy system was up to 3 times lower when using ActivFORMS comparing to regular coding of the system.

ActivFORMS considers goals as first-class citizens and is designed to support runtime updates of the feedback loop, which allows to dynamically change models of the adaptation components handling changing goals. Deploying small models enables focussed verification, which is important to deal with the state space problem inherent to verification. Furthermore, supporting dynamic updates is important to deal with uncertainty, in particular dynamically adding new goals. It is noteworthy to mention that runtime updates of the active model is a lightweight process.

ActivFORMS paves the way for several lines of future research. In our current work, we are extending the virtual machine to support probabilistic timed automata. The current version of ActivFORMS supports online detection of simple goal violations. We will enhance the support for more advanced goal models and study how to handle dependencies between goals. In the future, we also plan to study how we can introduce efficient runtime verification in ActivFORMS. Our aim is to enhance ActivFORMS with a plugin for incremental verification at runtime, which would allow verifying goals within a restricted time window. Another line of research that we plan to explore is supporting coordination between formal models. Currently, an active model interacts only with the local managed system and its environment. Supporting interactions between formal models would open both a way to handle multiple concerns locally and coordination between distributed active models in a decentralized setting.

## 7. REFERENCES


[1] G. Behrmann, R. David, and K. G. Larsen. A tutorial on uppaal. pages 200–236. Springer, 2004.

[2] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098, pages 87–124. Springer, 2004.

[3] G. Blair, N. Bencomo, and R. France. Models@ run.time. *Computer*, 42(10):22–27, Oct 2009.

[4] Y. Brun et al. Engineering self-adaptive systems through feedback loops. volume 5525 of *Lecture Notes in Computer Science*, 2009.

[5] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE TSE*, 37(3):387–409, May 2011.

[6] B. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems, LNCS vol. 5525*. 2009.

[7] R. de Lemos et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II, LNCS vol. 7475*. Springer, 2013.

[8] R. de Lemos, D. Garlan, and H. Giese. Software Engineering for Self-Adaptive Systems: Assurances, Dagstuhl Seminar 13511. 2013.

[9] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE*, 2009.

[10] N. Esfahani, E. Kouroshfar, and S. Malek. Taming uncertainty in self-adaptive software. In *ESEC/FSE*, 2011.

[11] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. ICSE, 2011.

[12] D. Garlan et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37:46–54, 2004.

[13] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. WOSS, 2002.

[14] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *ICSE*, 2013.

[15] D. Gil de la Iglesia and D. Weyns. Guaranteeing robustness in a mobile learning application using formally verified mape loops. SEAMS, 2013.

[16] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1), 2003.

[17] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, Nov. 1990.

[18] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. *FOSE*, 2007.

[19] V. P. L. Manna, J. Greenyer, C. Ghezzi, and C. Brenner. Formalizing correctness criteria of dynamic updates derived from specification changes. In *SEAMS*, pages 63–72, 2013.

[20] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *ICSE*, 1998.

[21] J. Tretmans. Formal methods and testing. chapter Model based testing with labelled transition systems, pages 1–38. Springer-Verlag, Berlin, Heidelberg, 2008.

[22] M. Usman Iftikhar and D. Weyns. A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System. *Foundations of Coordination Languages and Self Adaptation, FOCLASA, ArXiv e-prints*, Aug. 2012.

[23] T. Vogel and H. Giese. Model-driven engineering of self-adaptive software with eurema. *ACM Trans. Auton. Adapt. Syst.*, 8(4):18:1–18:33, Jan. 2014.

[24] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad. A survey of formal methods in self-adaptive systems. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pages 67–79, New York, NY, USA, 2012. ACM.

[25] D. Weyns, U. Iftikhar, and J. Soderland. Do External Feedback Loops Improve the Design of Self-Adaptive Systems? A Controlled Experiment. In *SEAMS*, 2013.

[26] D. Weyns, S. Malek, and J. Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.*, 2012.

[27] J. Zhang and B. Cheng. Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering*. ACM, 2006.

[28] J. Zhang, H. J. Goldsby, and B. H. Cheng. Modular verification of dynamically adaptive systems. AOSD, 2009.