Coordinated Actor Model of Self-adaptive Track-based Traffic Control Systems

Maryam Bagheri[a,*], Marjan Sirjani[b,c,*], Ehsan Khamespanah[d,c], Narges Khakpour[e], Ilge Akkaya[f], Ali Movaghar[a], Edward A. Lee[f]

[a] *Sharif University of Technology, Tehran, Iran*
[b] *Mälardalen University, Västerås, Sweden*
[c] *Reykjavik University, Reykjavik, Iceland*
[d] *Tehran University, Tehran, Iran*
[e] *Linnaeus University, Växjö Campus, Sweden*
[f] *University of California at Berkeley, California, U.S.A*

Abstract

Self-adaptation is a well-known technique to handle growing complexities of software systems, where a system autonomously adapts itself in response to changes in a dynamic and unpredictable environment. With the increasing need for developing self-adaptive systems, providing a model and an implementation platform to facilitate integration of adaptation mechanisms into the systems and assuring their safety and quality is crucial. In this paper, we target Track-based Traffic Control Systems (TTCSs) in which the traffic flows through pre-specified sub-tracks and is coordinated by a traffic controller. We introduce a coordinated actor model to design self-adaptive TTCSs and provide a general mapping between various TTCSs and the coordinated actor model. The coordinated actor model is extended to build large-scale self-adaptive TTCSs in a decentralized setting. We also discuss the benefits of using Ptolemy II as a framework for model-based development of large-scale self-adaptive systems that supports designing multiple hierarchical MAPE-K feedback loops interacting with each other. We propose a template based on the coordinated actor model to design a self-adaptive TTCS in Ptolemy II that can be instantiated for various TTCSs. We enhance the proposed template with a predictive adaptation feature. We illustrate applicability of the coordinated actor model and consequently the proposed template by designing two real-life case studies in the domains of air traffic control systems and railway traffic control systems in Ptolemy II.

Keywords: Self-adaptive Systems; Track-based Traffic Control Systems; Model@Runtime; MAPE-K Feedback Loop; Ptolemy II Framework

1. Introduction

Background. Traffic control systems are a wide range of systems that usually consist of a large number of moving objects whose movements should be coordinated and directed safely and efficiently, e.g. the air traffic control systems, railway systems, smart hubs, autonomous cars, maritime transportation, robotic systems, etc. Track-based Traffic Control Systems (TTCSs) are a class of traffic control systems that work based on a track-based design of the movement paths, i.e. traffic is passed through the pre-specified tracks and is directed by a controller. Based on the safe distance between the moving objects, each track is divided into several sub-tracks. The controller should be able to control the traffic in unforeseen situations by redirecting the traffic flow safely and efficiently, e.g. when a storm happens in some of the sub-tracks in an air traffic control system or some of the rails are frozen in winter conditions in a railway system.

*Corresponding authors
Email addresses: `mbagheri@ce.sharif.ir` (Maryam Bagheri), `marjan@ru.is` (Marjan Sirjani)

Two challenges of designing correct and efficient TTCSs, among others, are discussed as follows. Firstly, modern TTCSs are complex and highly distributed systems that operate in a dynamic changing environment. Designing such complex systems requires advanced techniques and supporting development tools that can deal with an uncertain environment and the system's complexities. Secondly, assuring safety (e.g. collision avoidance) and efficiency of the operations in such systems is indispensable and complicated. In particular, due to the dynamic nature of TTCSs, static analysis cannot solely be sufficient to ensure safety and quality in such systems, and we need online techniques to analyze the system at runtime as well.

To address the first challenge, we employed self-adaptation techniques and the Ptolemy II framework [1] to design and develop an autonomous Air Traffic Control System (ATC) in [2]. Self-adaptation is a technique to design complex systems that enables the system to autonomously adapt its structure and behavior in response to changes. The MAPE-K feedback loop is a common approach for realizing self-adaptation [3] that consists of *Monitor*, *Analyze*, *Plan*, and *Execute* components, orchestrated based on the cumulative information in the *Knowledge* component. Since the behavior of a self-adaptive system and subsequently its model change at runtime, a model of the system, called model@runtime, is maintained at runtime. A model@runtime is often an abstract model of the system and/or environment which is stored in the Knowledge component of the MAPE-K feedback loop and updated periodically at runtime. In the large-scale self-adaptive systems of systems, a system is usually designed with multiple feedback loops that interact with each other to achieve a goal collaboratively; e.g. [4] uses multiple interactive MAPE-K feedback loops along with models@runtime, where the MAPE-K feedback loops collaborate with each other by exchanging their knowledge.

Ptolemy II is an actor-oriented open source modeling framework that provides an extensive library of actors and models of computation (MoCs) for modeling and simulation of cyber-physical systems. A MoC, defining semantics of the interactions between the actors, is implemented in a *director* component. In the Ptolemy model of the ATC application in [2], we extended a director of Ptolemy II to include Analyze and Plan activities of the MAPE-K feedback loop. Furthermore, we used the actors of Ptolemy II to build a model@runtime of the ATC application. Ptolemy provides support for connecting a model to the physical world through the sensors and updating the model@runtime based on the observations from the real system. In addition, Ptolemy II offers a graphical design environment that aids in visualization of the system architecture.

Formal verification and validation is one of the common methods to tackle the second challenge, i.e. to ensure safety and quality in self-adaptive systems [5]. A number of works use static design-time verification [6, 7, 8, 9, 10], where the models of systems are developed using formal modeling languages and analyzed against correctness properties, prior to the development of the real systems. Formal verification at runtime has also been applied to self-adaptive systems [11, 12, 13, 14, 15, 16], where the current execution is examined against given properties.

Motivation. Design, analysis, and implementation of large-scale self-adaptive systems in general, and TTCSs specifically, require appropriate models and implementation platforms that can handle scalability issues and support runtime safety and performance analysis. Employing a formal and faithful model@runtime that reflects the real system behavior enables us to perform precise runtime analysis, as static analysis is not sufficient for safety assurance in self-adaptive systems. Furthermore, developing the MAPE-K feedback loop specially in large-scale self-adaptive systems needs appropriate platforms that support it to provide the required modularity and scalability. It is desirable that the employed models and implementation platforms provide support for predictive adaptations, i.e. to predict a fault prior to occurrence and perform a proper adaptation to adapt the system behavior. This is particularly crucial in safety-critical systems such as TTCSs. For instance, it is rational and cheaper to predict and prevent a collision in a TTCS than trying to recover the system from a collision. To the best of our knowledge, there is no integrated approach to design, analyze, and implement large-scale self-adaptive systems that is modular, explicitly models the MAPE-K feedback loops, uses models@runtime, provides facilities to perform predictive adaptations, and supports online analysis. The existing works either do not explicitly model the MAPE-K feedback loop [13, 14, 15, 16], do not employ model@runtime [12], do not support predictive adaptations [4, 12], or are not scalable [12]. We are not aware of such an approach in the domain of TTCSs either.

Lack of such an approach motivates us to propose a scalable model and an implementation platform for realizing self-adaptive TTCSs in this paper, based on our earlier work in [2]. The proposed model is aligned with real-world applications of TTCSs and supports a predictive adaptation mechanism. Furthermore, it is extended to support multiple interactive MAPE-K feedback loops. The proposed platform allows engineers to easily integrate self-adaptation mechanisms that are designed based on the MAPE-K feedback loops into software systems, use executable models@runtime to do safety and performance analysis, connect the models@runtime to the physical systems, and perform predictive adaptations by executing the models@runtime to predict some of the possible failures.

Contributions. In [2], we introduced a coordinated actor model to realize self-adaptive TTCSs. Furthermore, we depicted its applicability in building self-adaptive TTCSs by implementing and analyzing a case study in the domain of ATC in North Atlantic Organized Track System (NAT-OTS) [17] in Ptolemy II. The coordinated actor model encapsulates an actor-based model@runtime along with a coordinator. The coordinator is responsible for controlling the message passing among the actors, and contains the Analyze and Plan components of a MAPE-K feedback loop. The model@runtime is used by the coordinator to ensure satisfaction of safety properties and to adapt the system by predicting requirements violation, e.g. performance degradation.

In this paper, we extend our work in [2] by providing a formal description of the coordinated actor model using sequence and activity diagrams, illustrating usability of the coordinated actor model in realizing different applications of TTCSs, and describing the ability of the coordinated actor model to support predictive adaptation mechanisms. To show usability of the coordinated actor model, we describe how the coordinated actor model is completely matched with the nature of TTCSs. Toward this aim, we illustrate the mapping between different applications of self-adaptive TTCSs and the coordinated actor model, where each sub-track is modeled by an actor, the moving objects are modeled as messages passing among the actors, and the traffic controller is modeled by a coordinator. We also introduce multiple interactive coordinated actor models that promise scalability in design and analysis of self-adaptive TTCSs in a decentralized setting. In other words, one traffic controller cannot manage the whole traffic network in the large-scale TTCSs. Therefore, the traffic network is divided into multiple smaller control areas, where each area has its own traffic controller. We consider one coordinated actor model per each control area and provide multiple MAPE-K feedback loops interacting with each other. To support developing self-adaptive TTCSs using the coordinated actor model, we provide a template based on Ptolemy II that can be instantiated for constructing and analyzing various applications of self-adaptive TTCSs. In addition to ATC, a real-world application in the domain of railway traffic control systems is implemented by instantiating the developed template in Ptolemy II. We also implement a hierarchical model of ATC with several control areas that shows the interacting MAPE-K feedback loops. Moreover, we discuss how the features of Ptolemy II make it a suitable modeling framework to design and implement self-adaptive systems in the domain of cyber-physical systems in general and TTCSs specifically. Ptolemy II supports hierarchical modeling, which enables us to develop large-scale self-adaptive systems that consist of several MAPE-K feedback loops structured in different styles. In addition, support in Ptolemy II for connecting to models of the physical world enables us to perform our analysis based on more accurate data and consequently to be able to perform more effective adaptations. Furthermore, a Ptolemy model can be executed at runtime, which allows us to predict future behavior of the system and carry out suitable adaptations to avoid undesired behaviors.

The contributions of our work compared to [2] are summarized as follows.

1. We provide a formal description of the coordinated actor model.
2. We provide a general mapping between various TTCSs and the coordinated actor model.
3. We introduce multiple interactive coordinated actor models for developing multiple interactive MAPE-K feedback loops.
4. We discuss varying facilities of the Ptolemy II framework that make it suitable for modeling, simulation, and analysis of large-scale self-adaptive systems.
5. We extend the ATC Ptolemy model in [2] to become a generic Ptolemy template for modeling and analyzing different self-adaptive TTCSs.

6. We extend our Ptolemy template with the proposed predictive adaptation mechanism in the domain of ATC [2] to support adaptation of self-adaptive TTCSs based on predication.
7. We use our template to model a case study in the domain of railway systems.
8. We implement a hierarchical model of ATC in Ptolemy II to capture the decentralized nature of ATC over different control areas. The support for hierarchical modeling in Ptolemy II makes this framework and our template scalable for modeling decentralized systems or large-scale systems of systems.

The rest of the paper is organized as follows: We provide a general overview of TTCSs as our problem domain and describe their main design requirements in Section 2. Section 3 presents the coordinated actor model, multiple interactive coordinated actor models, and general mapping between various TTCSs and the coordinated actor model. In Section 4, we introduce Ptolemy II and explain its advantages for developing self-adaptive systems. Furthermore, we demonstrate our Ptolemy template for developing the MAPE-K feedback loops along with the model@runtime of self-adaptive TTCSs. To illustrate how the coordinated actor model and our proposed Ptolemy template are used for developing self-adaptive TTCSs, we describe different adaptation policies for a case study in the domain of ATC and compare their performance in Section 5. Section 6 discusses state of the art approaches in domain of the self-adaptive systems. We conclude the paper and outline the future work in Section 7.

2. Track-Based Traffic Control Systems (TTCSs)

In this section, we provide an overview of the autonomous TTCSs by introducing air traffic control system and rail traffic control system as two large-scale safety-critical instances of TTCSs, and describe two adaptation levels for designing autonomous TTCSs. Furthermore, we outline design requirements of the self-adaptive TTCSs.

2.1. Overview

Transportation systems are large-scale cyber-physical systems that besides organized elements and their interactions, consist of a traffic demand which should be safely transported from one place to another place. One class of transportation systems is track-based transportation systems (TBTS), in which the traveling space is divided into smaller safe regions to reduce the risk of collision between moving objects. These regions are called tracks which, based on the safe distance between the moving objects, are divided into several sub-tracks. In TBTS, traffic flows through certain sub-tracks and is directed by a centralized traffic control system. A traffic control system uses supervision instruments to monitor and control movements of the objects. Air and rail transportation systems are widely used transportation systems [18, 19, 20]. These systems are usually constructed based on a track-based infrastructure. The air traffic control system (ATC) in North Atlantic is based on an Organized Track System (OTS) that follows the track-based structure [17]. Similarly, the same pattern appears in rail traffic control systems. In both systems, sub-tracks are critical sections, accommodating only one moving object in-transit.

In traffic control systems, the traffic controller manages flow of the traffic and assures safe movement of the moving objects. In order to have a fully automated traffic control system, sub-tracks do not have decision making abilities and the controller is responsible for managing the traffic, taking into account the environmental changes and congestion conditions. For example, the controller is able to accept or reject a moving object to enter into a sub-track, reschedule or reroute the moving objects and so forth. Air and rail transportation systems have centralized traffic control systems. A controller knows structure of the traffic network and upon any change in the system, updates its knowledge about the current traffic.

A part of the Tokyo subway route map, available in [21], is shown in Fig. 1. Subway systems have a similar structure with other rail transportation systems such as the railway systems. As shown in Fig. 1, the subway lines that consist of stations and junctions are the main elements of rail traffic control systems. The sub-tracks in subway systems are called blocks. The distance between two stations is divided into several blocks. The stations in Fig. 1, are shown with the round-cornered rectangles. The junction is a place where the passengers change their traveling lines and consists of several stations, e.g. junction P01 contains two stations M06 and E30. Each station accommodates at most one train at each time. A station at a junction

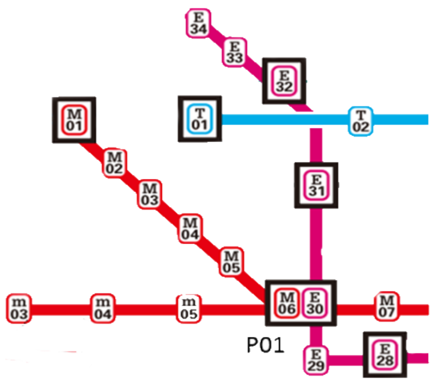


Figure 1: A part of the Tokyo subway route map [21]. Different lines of traveling are shown in different colors. Stations and junctions are shown with colorful and black rectangles, respectively.

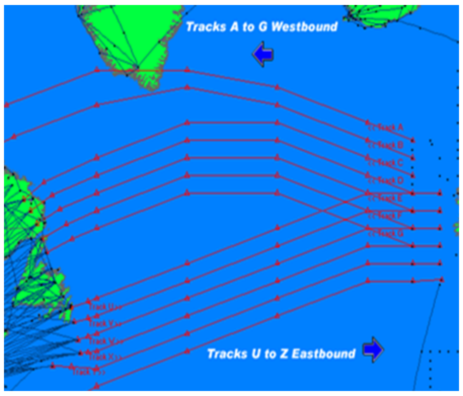


Figure 2: The infrastructure of NA-OTS [22]. The tracks, divided into several sub-tracks, are shown by the red lines. The aircraft in a sub-track can be navigated toward a sub-track in north, south, or west/east.

might have more than one entrance or exit for the trains. The source (destination) of the traffic in the rail traffic control system is a station with one or several exits (entrances).

The infrastructure of North Atlantic OTS (NA-OTS), available in [22], is shown in Fig. 2. Each red line shows a track. The westbound tracks navigate the aircraft from Europe to north America and Canada and the eastbound tracks navigate the aircraft from north America to Europe. The flights across the Atlantic Ocean are controlled by the oceanic control centers. The triangles in Fig. 2, are the oceanic waypoints, where the tracks are divided into the sub-tracks. The aircraft travel through the sub-tracks toward one of the three directions north, south, and west/east. For instance, each aircraft of the eastbound traffic is able to travel toward a sub-track in north, south, and east. The aircraft can fly from different exits of a source airport and arrive at different entrances of a destination airport.

Transportation systems are sensitive to internal failures and environmental changes. A disturbance in a railway system can lead to delays and consequently causes conflicts among the schedules of the trains. In such circumstances, the delays may spread over the whole network and affect the overall cost and satisfaction of the passengers. To recover from these disturbances, it is required to reschedule the timetable of the trains, the assigned routes to the trains, the drivers, etc. Similarly, any undesired and unforeseen changes in the air transportation systems such as dynamic changes in the weather conditions, delays in landing and taxiing, strikes of the staffs, and failures of the aircraft components will require re-planning the flight plans of the aircraft. A flight plan includes the aircraft's flight route, fuel, speed, etc.

In such systems, the ultimate goal is to minimize the delay and cost, and to this end, suitable adaptations should be safely performed in the system, e.g. finding a feasible schedule for the trains, assigning new flight

Table 1: Two levels of adaptation in TTCSs

Adaptation Level	**Adaptation in TTCSs**
First Level	Changing the route/ schedule or speed of moving objects
Second Level	Switching between different rerouting/ rescheduling algorithms

plans to the aircraft, etc. In addition, as the transportation systems are safety-critical systems, several safety concerns must be carefully considered when designing an adaptation. For instance, the adaptation should be designed in a way that avoids a collision and considers the fuel level required for the new schedule/flight plan. We call all systems that follow the track-based structure described above and have the ability to adapt themselves to the changes, as self-adaptive TTCSs. For instance, a network on chip (NOC) can be considered as a TTCS in which packets, as the moving objects, travel through a set of sub-tracks (routers). However, unlike air and rail traffic control systems, a NoC does not have a centralized controller and its traffic control algorithm is distributed among the routers [23].

Self-adaptive TTCSs support behavioral adaptation; i.e. behavior of the system is modified by changing its parameters or switching between different algorithms. Self-adaptive TTCSs can be designed with two levels of adaptation. As shown in Table. 1, in the first level of adaptation, the controller uses a default rerouting/rescheduling algorithm to assign new routes/schedules to the moving objects or to change their speeds. For instance, consider that a storm happens in a part of the airspace. The controller of ATC uses a rerouting algorithm to assign new routes to the aircraft whose initial routes cross over the sub-tracks affected by the storm. In an advanced example, consider that an aircraft affected by the storm is rerouted and its new route inevitably crosses over a sub-track that previously has been reserved for passage of a second aircraft at the time. In this circumstances, the controller can change speeds of the both aircraft to avoid the conflict. In the second level, future behavior of the system under executing different rerouting/rescheduling algorithms is predicted. Then, based on the provided results such as values of the performance metrics, the controller is adapted by switching between different rerouting/ rescheduling algorithms. For instance, consider that the controller of ATC uses a default rerouting algorithm to find the new routes for the affected aircraft by the storm. If it cannot find the new routes such that a certain property, e.g. safety, is guaranteed, it switches to another rerouting algorithm. However, the structure of the system, i.e. number of the sub-tracks and their interconnections, is fixed.

2.2. Design Requirements of Self-Adaptive TTCSs

There is a vast literature on challenges in developing self-adaptive systems, ranging from modeling [24], architecture and design [25, 26], goal and requirement engineering [24, 27, 28], adaptation in decentralized setting [29, 27], to assurance [24, 30], practical verification and validation [27, 31, 29], etc. Intuitively, similar challenges might appear in developing self-adaptive TTCSs. As discussed below, this paper focuses on the decentralized adaptation and assurance challenges, which are also considered as the primary requirements of self-adaptive TTCSs.

Decentralized adaptation. One of the characteristics of self-adaptive systems is the degree of decentralization [29]. The decentralization in self-adaptive systems deals with coordinating a control decision activity on different components [32]. For instance, a MAPE-K activity can be decentralized on different components that are coordinated through a coordination mechanism. Furthermore, a self-adaptive system of systems, consisting of autonomous systems, can be realized through coordination of multiple feedback loops (i.e. multiple MAPE-K feedback loops), in which each loop controls an autonomous system. Although decentralization improves scalability of large-scale self-adaptive systems, it is an important challenge in developing self-adaptive systems [29]. In other words, decentralized approaches need mechanisms for coordinating different adaptation activities/loops and system-wide assurance.

Self-adaptive TTCSs are large-scale systems in which adaptation cannot be handled by a single feedback loop. Developing interacting feedback loops is a main design concern for self-adaptive TTCSs. TTCSs are

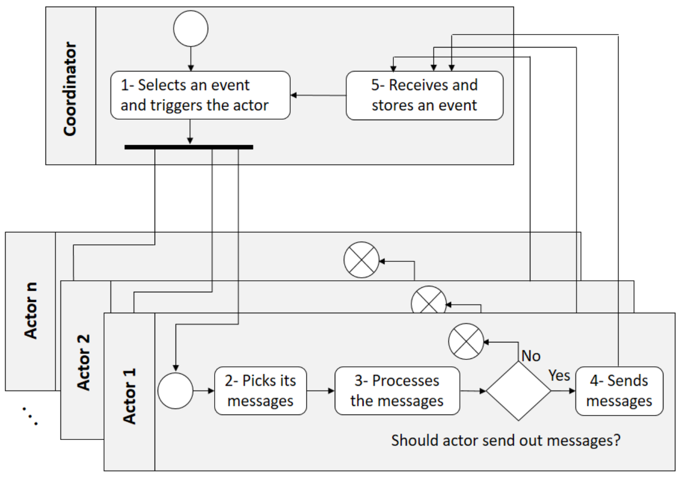

Coordinator
1- Selects an event and triggers the actor
5- Receives and stores an event
Actor n
Actor 2
Actor 1
2- Picks its messages
3- Processes the messages
No
Yes
4- Sends messages
Should actor send out messages?

Figure 3: The behavior of the coordinated actor model

divided into smaller control areas and each area is controlled through a separate traffic controller. The controllers have to interact with each other to satisfy objectives of the whole system. Modeling interactions between control areas is essential to forming a more complete model [28].

Assurance. Providing evidence that a system satisfies its requirements during its operation and augmenting the system with mechanisms to predict and avoid requirement violations, specially in safety-critical systems, are crucial issues. The pioneering solutions use model@runtime to detect or predict violation of the system requirements at runtime. However, the models should correctly reflect what can be seen and what can be affected [26]. Furthermore, exploiting the model@runtime and/or prediction mechanisms without employing suitable frameworks are complicated tasks [29].

As mentioned in Section 2.1, TTCSs are highly dynamic distributed systems with several safety concerns, e.g. safe movement of the moving objects, and performance objectives such as minimizing the delays. Because of the dynamic environment of TTCSs, satisfaction of these requirements should be guaranteed at runtime.

3. The Coordinated Actor Model for Self-Adaptive Track-based Traffic Control Systems

In this section, we provide an overview of the coordinated actor model and introduce multiple interactive coordinated actor models for developing decentralized self-adaptive systems. Then, we describe how a self-adaptive TTCS is realized through the coordinated actor model. Moreover, the mapping between a coordinated actor model and three different applications of self-adaptive TTCSs is explained. These applications are ATCs, rail traffic control systems, and centralized robotic systems.

3.1. A Coordinated Actor Model

We introduced the coordinated actor model, consisting of a coordinator and a set of actors, in [2]. The behavior of a coordinated actor model is illustrated by an activity diagram in Fig. 3. Actors communicate via asynchronous message passing. The coordinator includes a scheduler that governs message passing among the actors. For each message communicated between the actors, an event is triggered for the scheduler. The scheduler receives the event and puts it into its internal buffer (activity 5 in Fig. 3). Then, it selects an event from the internal buffer based on a policy and triggers the receiver actor (activity 1). The policy of the scheduler in a timed system is usually selecting an event with the least arrival time. The triggered actor picks and processes its received messages (activities 2 and 3). It also can send several messages to other actors (activity 4).

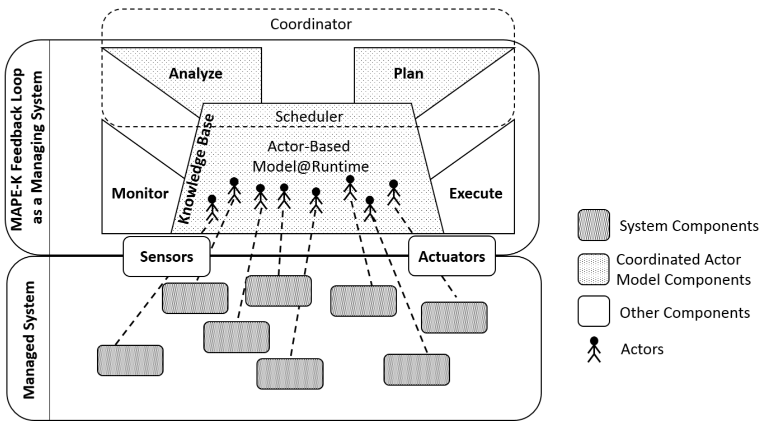


Figure 4: Modeling self-adaptive systems with the coordinated actor model

The coordinated actor model is used to build the model@runtime and Analyze and Plan activities of a MAPE-K feedback loop. The model@runtime is stored in the Knowledge component of the MAPE-K feedback loop, and upon any system or environmental changes, is updated through the Monitor component. Then, it is analyzed by the Analyze component, and based on the analysis results, the Plan component makes a decision for adapting the system. The decision is sent to the system through the Execute component. The model@runtime in a coordinated actor model encloses actors and the scheduler part of the coordinator. The coordinator includes a decision maker. The decision maker performs the Analyze and Plan activities of the MAPE-K feedback loop. The mapping between a coordinated actor model and model@runtime, Analyze, and Plan activities of a MAPE-K feedback loop is illustrated in Fig. 4.

In order to show the behavior of a coordinated actor model for realizing a MAPE-K feedback loop, we depict a scenario of the interactions between different elements of the coordinated actor model along with the Monitor and Execute components by the sequence diagram in Fig. 5. Suppose that a failure happens in a system whose components are modeled by the actors. The actor-based model@runtime is updated by the Monitor component (arrow 1 in Fig. 5, which is interpreted as a message in the sequence diagram). The actors inform the coordinator about their current situations (message 2). This way, the coordinator obtains new information about the model@runtime (message 3). Moreover, since the events of the scheduler have references to the receiver actors, the Analyze and Plan activities can acquire necessary information from the model@runtime using the internal buffer of the scheduler (messages 5 and 8). The analyzer detects the failure as a special change in the system (message 4). By detecting the change, the planner is activated (message 6), obtains its needed information (messages 7 and 8), and makes a plan for adapting the system (message 9). The model@runtime is updated by the designed adaptation (message 10). The designed adaptation can be blindly issued to the system or the model@runtime can be analyzed to obtain more information (message 11). For instance, the actor-based model@runtime can be checked against the system requirements and in case of violation, other adaptation decisions are investigated (loop 1). Otherwise (else part of alt 1 fragment), the adaptation is finalized (message 12) and is issued to the Execute component (message 13). It is noteworthy that the activation, activated by message 9, checks the model@runtime, and this way the same behavior as Fig. 3 appears.

3.2. Multiple Interactive Coordinated Actor Models

There are several approaches to coordinate multiple feedback loops in a decentralized setting. For instance, the feedback loops could collaborate with each other by sharing their knowledge or they could be organized in a hierarchical way, while a layer enabling their interactions is added on top of them. Based on the latter case, we introduce multiple interactive coordinated actor models for achieving self-adaptation in a decentralized setting, where adapting a large-scale system cannot be handled by a centralized feedback loop. As shown in Fig. 6, multiple interactive coordinated actor models consist of a coordinated actor model

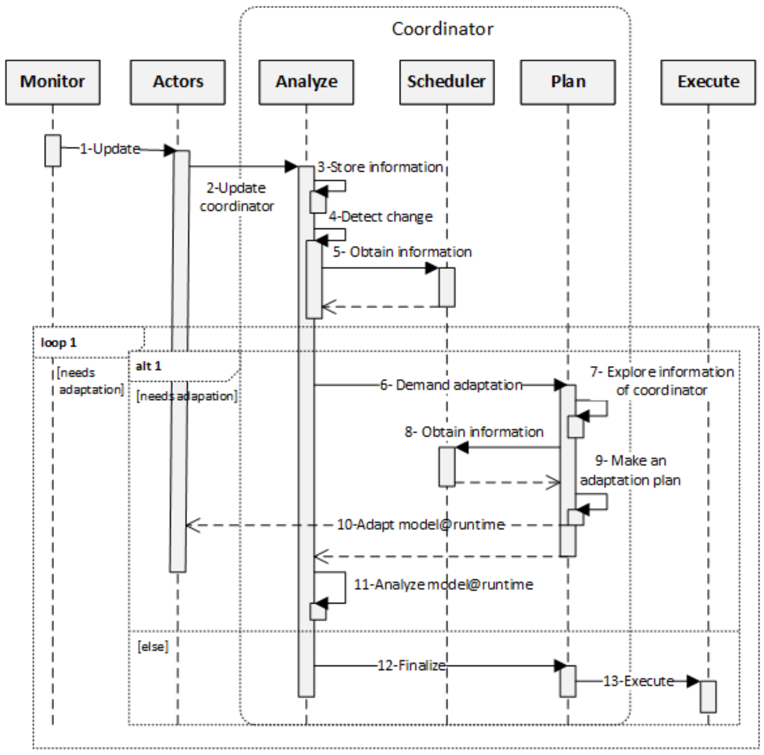


Figure 5: A simple scenario of the interactions between different elements of a coordinated actor model along with the Monitor and Execute components

per each subsystem and a top-level coordinator. Each coordinated actor model has its own model@runtime and keeps an actor model of a subsystem in its knowledge component. The interactions between different coordinated actor models are managed by a top-level coordinator, which contains an abstract overview of each lower-level coordinator. The message passing between the actors in two coordinated actor models is governed by the scheduler of the top-level coordinator. The lower-level coordinators are able to inform the top-level coordinator about their current situations or request more information. For instance, the Plan activity of a lower-level coordinator can request information about other subsystems from the top-level coordinator. Furthermore, the top-level coordinator has access to the lower-level coordinators. For instance, the Analyze part of the top-level coordinator can request local analysis results from the Analyze parts of the lower-level coordinators.

Employing multiple interactive coordinated actor models promises scalability in design (modular design) and analysis (modular analysis). For instance, instead of analyzing a large-scale system, the subsystem affected by a change is analyzed using a compositional reasoning technique. A simple scenario of the interactions between different elements of multiple interactive coordinated actor models along with the Monitor and Execute components is illustrated in the sequence diagram in Fig. 7. In this figure, we suppose that the model consists of two coordinated actor models, coordinated_actor_model_1 and coordinated_actor_model_2, whose models@runtime are updated by the Monitor component. Furthermore, we suppose that the change happens to the associated subsystem with coordinated_actor_model_1. As shown in Fig. 7, the coordinator of coordinated_actor_model_1 requests information from the top-level coordinator and analyzes its model@runtime separately (using compositional reasoning techniques, i.e. compositional verification [33, 34]). This way, modular analysis and modular adaptation (adapting a part of the system) are provided.

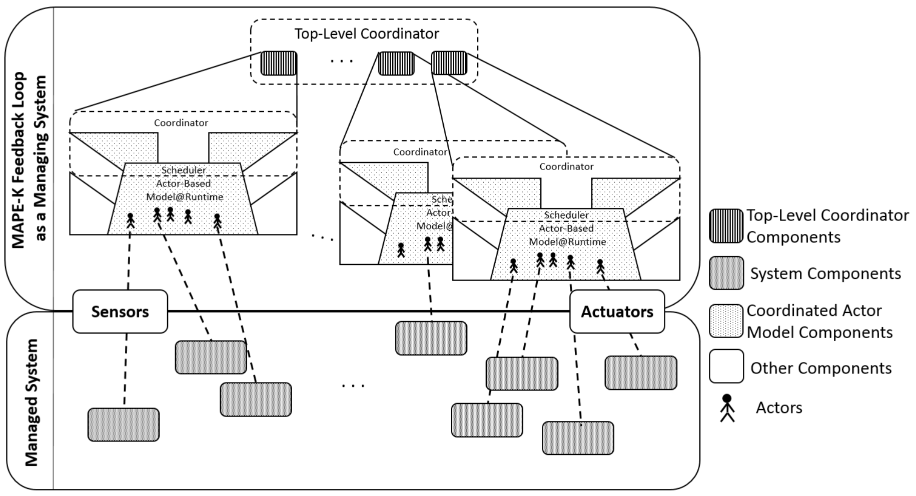

MAPE-K Feedback Loop as a Managing System
Managed System
Top-Level Coordinator
Coordinator
Scheduler
Actor-Based Model@Runtime
Sensors
Actuators
Top-Level Coordinator Components
System Components
Coordinated Actor Model Components
Other Components
Actors

Figure 6: Decentralized self-adaptation using multiple interactive coordinated actor models

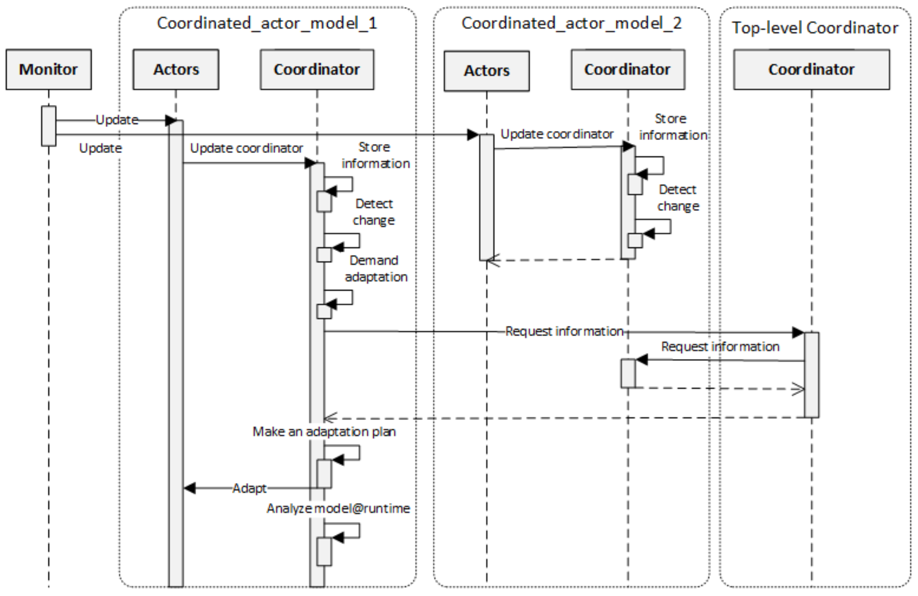


Figure 7: The behavior of multiple interactive coordinated actor models along with the Monitor and Execute components in a simple scenario

3.3. The Mapping between Self-Adaptive TTCSs and the Coordinated Actor Model

A self-adaptive TTCS consists of a set of nodes (i.e. sub-tracks, sources and destinations), forming a mesh in which a traffic flow passes through the nodes. Each moving object carries the necessary information, i.e. traveling route, fuel, speed, identification number, and arrival time at each node of the route. Each node is considered a critical section and accepts a limited number of moving objects. This limitation is called the capacity of the node. In a full node, extra incoming moving objects are prevented from entering into the node. The traveling time of a moving object toward the next node is calculated based on its speed and the distance to its next node.

There are three different types of nodes in a traffic network; source nodes, destination nodes, and mid-nodes. A source node is a place where moving objects start their journeys. A destination node is the target of travels. The node which is placed inside a mesh is called mid-node. A block in a rail traffic control system is a mid-node with a single entrance and exit. If a node has more than one entrance, it might receive several moving objects simultaneously. In this case, the controller uses a pre-specified policy to accept a number of moving objects based on the node's remaining capacity. For instance, in an air traffic control system, an aircraft with the less remaining fuel has priority for entering into a sub-track. In the case of simultaneous arrivals of the moving objects at different entrances of a node, collision between the moving objects is inevitable. To prevent collision, each moving object announces its request to enter into a node to the controller in a predetermined time before its arrival at the node. Then, the controller decides on accepting or rejecting the moving object. The node is reserved for the moving object which is accepted for entering into it. The rejected moving object is rerouted if the node, accommodating it, has multiple exits. Finding the route, and rerouting or rescheduling the moving objects are performed by the controller. A source (destination) node of the traffic flow has the determined capacity for departure (arrival) of the moving objects at the same time. Therefore, the moving objects in a source node might have delays based on the capacity of the node for departure. The moving objects, arriving at a destination node at the same time, might have delays based on the capacity of the destination node for simultaneous arrivals.

To model self-adaptive TTCSs using the coordinated actor model, an actor is associated with each node and the moving objects are modeled as messages passed through the actors. Each message carries necessary information, i.e. traveling route, fuel, speed, etc. When a message is passing through an actor, the content of the message and also the internal state of the actor are changed. Furthermore, the effect of the environment on a node is reflected in the state of the associated actor with the node. A node is available, and consequently a moving object can travel through it, if it is not affected by an adversarial environmental condition or it is not occupied or reserved by another moving object. Otherwise, the node is unavailable. Therefore, the effect of environment on a node is reflected on the associated actor with the node by switching its state between available and unavailable. In TTCSs, the controller is modeled by a coordinator. If the state of the actor is changed, the actor informs the coordinator, and this way, the coordinator obtains a complete map of the mesh and details of the current traffic. In response, the coordinator exploits different adaptation policies and affects on the system behavior. For instance, in the first level of adaptation, the coordinator can use a rerouting algorithm to find the new routes for the moving objects. The route of a moving object is modeled as a sequence of messages that are sent by the adjacent sub-track actors.

The activity diagram in Fig. 3 is adjusted to illustrate the overall behavior of the coordinated actor model of the self-adaptive TTCSs. The new activity diagram is depicted in Fig. 8. It has four activities other than the primary activities of the activity diagram in Fig. 3. Activities 1, 2, and 3 are added for adaptation purpose and activity 9 shows the response of the coordinator to a request by an actor. The activity diagram in Fig. 8 is initialized when a change (e.g. a storm in the airspace) happens to the environment of a TTCS. Then, the Monitor component updates states of the actors (activity 1). The state of each actor explains whether the corresponding node to the actor is affected by the change, or it has been occupied by a moving object. The state also keeps some information about the moving object, such as its speed, the traveled distance by the moving object, etc. Each actor informs the coordinator about its current state (activity 2). This way, information in the coordinator about the traffic network is updated. For instance, assuming that the coordinator knows the topology of the traffic network, it exploits current positions of the moving objects in the traffic network. The topology of the traffic network is determined by extracting the connections

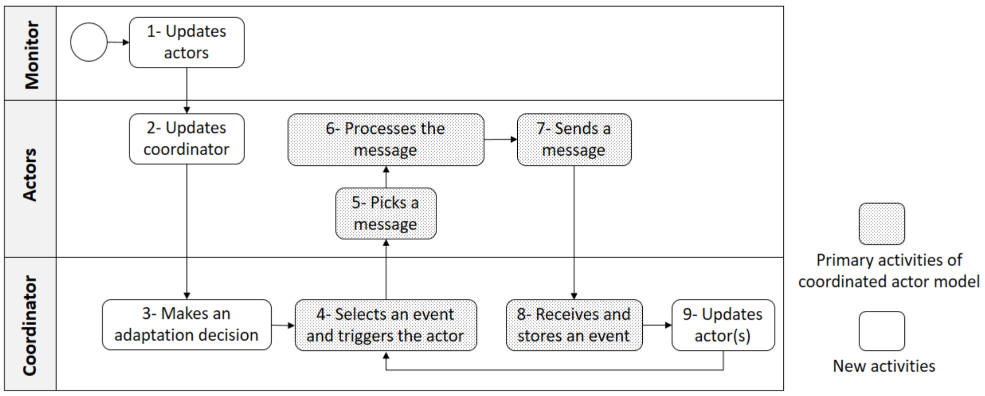


Figure 8: The behavior of the coordinated actor model of a self-adaptive TTCS. Primary activities of the coordinated actor model are mapped into the illustrated activities in Fig. 3.

between the actors. The coordinator uses its information, makes an adaptation decision, and issues the designed adaptation to the actors (activity 3). To this end, the coordinator removes or updates messages of the actors or creates new messages if needed. After adaptation, each message contains a new traveling plan for the moving object traveling through the corresponding node to the actor. To figure out whether the issued adaptation in response to an environmental change leads to violating a specific property, the model@runtime is checked against the property. Toward this aim, the model@runtime reflects the behavior of a real-world traffic control system in which the moving objects travel toward their destinations. By firing the actors, movements of the moving objects along with the time they need for traveling or requesting new decisions from the controller are simulated. The coordinator uses the policy of its scheduler part to select an actor for firing (activity 4). The actor receives a message when it is fired (activity 5). This message models the moving object arriving at the corresponding node to the actor. The actor also changes its state according to the message (activity 6). For instance, it marks itself as an occupied actor and stores information about the moving object. We assume that the moving objects in an automated TTCS are navigated by a centralized controller and need its decisions in different situations. For instance, when a moving object enters into a node, it might request new information from the controller for navigation purpose. Toward this aim, the actor sends its received message to itself (activity 7). The coordinator is informed by receiving an event (activity 8), decides about the demand, and issues its decision to the requesting actor (activity 9). For instance, the coordinator can update the traveling plan of the moving object by changing the message that will be received by the actor. The activities 4 to 9 are repeated several times for different purposes. The actor is fired again (activity 4) and receives an updated message in which the traveling direction toward the next node is given (activity 5). The actor processes the message for a while (activity 6). This way, the traveling time of the moving object along the corresponding node to the actor is simulated. It is assumed that each moving object requires permission of the controller for entering into the next node in its route. To this end, the actor ready to send a message (the moving object is leaving its current node) informs the coordinator about its decision to send a message. It first sends a message to the actor corresponding to the next node in its route (activity 7). The coordinator is informed by receiving an event (activity 8), decides about the request, and issues its decision to the actor (activity 9). For instance, a controller in the real-world, based on the situation of the next node, asks the moving object to decrease its speed before entering into the next node. To model this condition, the coordinator removes the sent message and creates a new message for the requesting actor. The actor is fired, receives its message, and processes the message for a while (decreasing speed is modeled by increasing computation time of the actor). If the actor is allowed, it sends its message to another actor (activity 7). This means that the moving object leaves its current node toward the next node in its route.

In our approach, following an occurrence of a change in the environment of a self-adaptive TTCS, an adaptation process is run in reaction to the change. The change in TTCSs does not necessarily cause an immediate disruption. It can be an event that may lead to a disruption in the future. Therefore, the adaptation in our approach can be performed to recover from a disruption or prevent possible future failures. In the latter case, the system is adapted before an environmental change (an event) leads to a problem; i.e. a moving object is rerouted before it enters into the sub-track affected by an adversarial environmental change. Besides responding to the change by an immediate adaptation, the coordinator is able to predict the subsequent behavior of the system. The coordinator is then able to select an adaptation policy that optimizes several performance metrics or prevents violation of the system requirements, i.e. preventing conflicts among the moving objects, a traffic blockage or deadlock, and performance degradation. Calculating the performance metrics or detecting the possible failures can be performed through model checking or simulation, since the coordinator, due to its access to the actors (through the events in the buffer of the scheduler), can execute the model@runtime and check it against the given requirements. Based on the above discussion, the coordinated actor model supports predictive adaptation, since the coordinator is able to obtain the future behavior of the system.

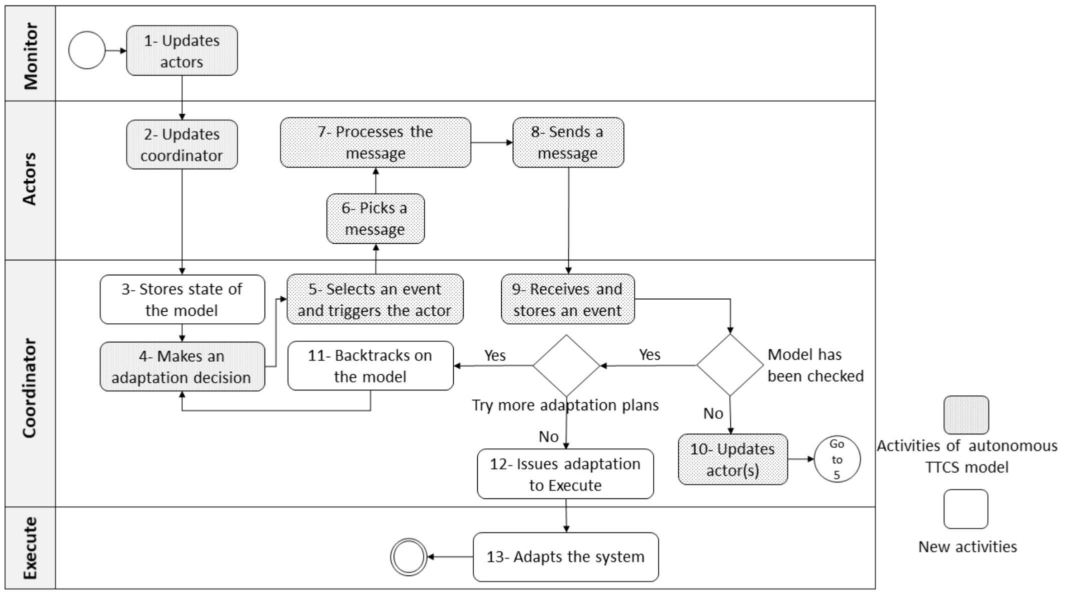
We add more details to the activity diagram in Fig. 8 to show how the coordinated actor model supports predictive adaptation. The resulted activity diagram is illustrated in Fig. 9. We assume that the coordinator has several adaptation policies. When the coordinator is informed about a change (e.g. an adversarial condition in a sub-track), the model@runtime enters into the prediction mode. In this mode, the coordinator takes a snapshot from the model@runtime at the change point (activity 3 in Fig. 9) and adapts the model@runtime to a new configuration. Then, the analyzer looks ahead through the model@runtime. This means that the model@runtime proceeds with its execution to the end (or a look-ahead horizon) (activities 5 to 10). During the execution, performance metrics are calculated and safety properties are checked. Based on the provided results, the planner decides to adapt the system or to try another adaptation policy to choose the most suitable one for adapting the system. In the latter case, the coordinator backtracks on the model@runtime to restore the information at the change point (activity 11) and starts a new look ahead iteration.

TTCSs are usually large-scale safety-critical systems in which adaptations cannot be handled by a centralized controller. These systems are divided into several control regions and each region has its own traffic controller. Therefore, a large-scale self-adaptive TTCS is modeled by multiple interactive coordinated actor models, a coordinated actor model per each control region. Each coordinated actor model follows the described behavior in Fig. 8 (or Fig. 9). The interactions between different control regions such as transporting a moving object from one control region to another control region is modeled by the top-level coordinator.

3.4. The Mapping between Real-World Applications of Self-Adaptive TTCSs and the Coordinated Actor Model

To illustrate the usability of the coordinated actor models in realizing self-adaptive TTCSs, we describe how ATCs, rail traffic control systems, and centralized robotic systems as real-world applications of self-adaptive TTCSs are mapped into the coordinated actor model. Furthermore, we explain the activities of Fig. 9, in which these three different applications have different implementations.

Robotic system. A robotic system involves a group of robots navigating in a shared workspace to achieve tasks in a coordinated fashion. We consider a workspace with a grid structure. In other words, the workspace consists of cube-shaped blocks. Each robot starts its journey from a starting location toward a destination location and has a route as a sequence of blocks traveled by the robot. The robots are placed in a dynamic environment in which new obstacles might be added to the workspace. A critical issue in such robotic systems is finding suitable collision-free paths for the robots while the environment is changing. There are different strategies for developing robotic systems such as centralized and decentralized approaches [35, 36]. In decentralized approaches, each robot finds its traveling route itself. We focus on centralized approaches, where a controller that can be a master robot instructs other robots [35]. Based on the pattern described in Section 2.1, a robotic system with a centralized controller is a TTCS. In such a robotic system, each block of the workspace is considered a sub-track accommodating only one robot at a time. The mapping between a centralized robotic system and a coordinated actor model is defined as follows.



Monitor
Actors
Coordinator
Execute
1- Updates actors
2- Updates coordinator
7- Processes the message
8- Sends a message
6- Picks a message
3- Stores state of the model
5- Selects an event and triggers the actor
9- Receives and stores an event
4- Makes an adaptation decision
11- Backtracks on the model
Yes
Yes
Model has been checked
Try more adaptation plans
No
No
10- Updates actor(s)
Go to 5
12- Issues adaptation to Execute
13- Adapts the system
Activities of autonomous TTCS model
New activities

Figure 9: The predictive adaptation for TTCSs using the coordinated actor model. Activities of the autonomous TTCS model are the illustrated activities in Fig. 8.

- Each robot is modeled as a message.
- The blocks of the workspace are modeled by actors.
- The controller (or the part of the master robot that has responsibility for controlling the robots) is modeled by a coordinator.

Air traffic control systems. As explained in Section 2.1, ATC consists of a set of tracks. Based on the safe distance between the moving objects, each track is divided into a set of sub-tracks. A sub-track is a critical section that accommodates only one moving object in-transit. The aircraft start their journeys from a source airport and travel through a sequence of sub-tracks toward their destination airports. In ATC, the controller is in charge of adapting flight plans of the aircraft, considering congestion and environmental changes. The mapping between an air traffic control system and a coordinated actor model is defined as follows.

- Each aircraft is modeled as a message.
- The sub-tracks, source airports, and destination airports are modeled by actors.
- The controller is modeled by a coordinator.

Rail traffic control system. As explained in Section 2.1, rail lines are the main elements of a rail traffic control system. Each rail line consists of a set of stations. The distance between two stations is divided into a set of blocks and each block is a critical section. The controller in the rail traffic control system is responsible for altering the time table of the trains or changing their routes such that no conflict happens between the trains. The mapping between a rail traffic control system and a coordinated actor model is defined as follows.

- Each train is modeled as a message.
- The blocks and stations are modeled by actors.
- The controller is modeled by a coordinator.

We are able to define the similar mappings for different applications of the self-adaptive TTCSs such as maritime transportation, unmanned vehicles, smart hubs, etc., in which ships and vehicles are modeled as messages, terminals and stations are modeled by the actors, and controllers are modeled by the coordinators. However, these applications might have different implementations for different activities of the activity diagram in Fig. 9. The rail traffic control systems, ATCs, and centralized robotic systems might have different adaptation policies (different implementations for activity 4 of Fig. 9). For instance, the controller in ATC uses a rerouting algorithm to reroute the aircraft intending to cross over sub-tracks affected by a storm, the controller in a rail traffic control system stops trains intending to cross over a broken block, and the controller in a robotic system, in the presence of a new obstacle, finds new routes for all robots of the system such that no conflict happens between them. Although, it is possible that the controller in an ATC or rail traffic control system changes traveling plans of all moving objects, these systems are large-scale and a global adaptation might impose high cost on the system. These applications might have little technical differences in other respects. The actors can ask coordinator whether they are allowed to send their messages. For instance, a/an train/aircraft asks the controller for permission when it is about to leave its current node. In response, the controller asks the train/aircraft to reduce its speed. In robotic systems, the robot does not need to ask the controller for permission.

4. Ptolemy II for Designing Self-Adaptive TTCSs

In this section, we introduce the Ptolemy II framework, discuss its advantages for modeling self-adaptive systems, and propose a Ptolemy template for modeling self-adaptive TTCSs. Our Ptolemy template is based on the coordinated actor model.

4.1. Ptolemy II Overview

Ptolemy II [1] is an actor-oriented open-source modeling framework. A Ptolemy model consists of actors that communicate via ports by message passing. Actors are implemented in Java with a well-defined interface, which includes ports and parameters. Ports are the communication points of actors and parameters are used to configure their behaviors. Each actor has a set of variables that determine its internal state. It also has a set of public methods which are called action methods. The action method *initialize* initializes state variables of the actor. The main computation of the actor is defined in the action method *fire*. This method reads input values from input ports, performs computation that can be dependent on parameter values, and generates outputs. In Ptolemy II, new actors are defined by extending or modifying the existing actors.

In Ptolemy II, the semantics of interactions and communications between actors is defined by a Model of Computation (MoC), implemented by a *director* component. Ptolemy II implements a library of MoCs, including Dataflow, Process Networks, Rendezvous, Finite-State Machines, Synchronous-Reactive, etc., which can be composed to yield heterogeneous models. The framework also provides a specialized actor, called a *Composite actor*, which enables building hierarchical models. As each composite actor is itself a Ptolemy model, it has its own MoC, which can be inherited from the container or can be defined within the composite actor itself. These features make Ptolemy II a suitable framework for modeling complex heterogeneous systems. Each subsystem within a complex system can be modeled within its own Ptolemy composite actor, where the semantics of internal actors is provided by a local director implementing a suitable MoC. One of the most widely used MoCs in Ptolemy II is Discrete Event (DE), in which each message communicated between actors has a time stamp. The DE director maintains a global event queue, called *eventQueue*, in which events are sorted based on their time stamps. Actors are triggered by the DE director in time-stamped order to process their messages. Each actor can send messages to its connected actors through the ports. The connections between the actors are instances of a channel entity which is defined by the director. The channels pass messages from one port to another port. A channel has a *put* function that is called when a message is sent over the channel. For each communicated message, the *put* function generates a time-stamped event for the director. Also, each actor can execute the *fireAt(t)* instruction, signaling the director to trigger the actor at some future time *t*. This can be used to simulate a time delay from inputs to outputs. It also is possible to define public functions in the director of Ptolemy II and invoke them in actors.

Ptolemy II is open source, and new MoCs can be developed by extending or modifying the existing MoC implementations. Heterogeneous hierarchical actor-oriented design capabilities in Ptolemy II provide a strong modeling and simulation toolkit for designing cyber-physical systems [1].

4.2. Advantages of Ptolemy for Modeling Self-Adaptive Systems

Ptolemy II is a powerful framework for modeling and analyzing cyber-physical systems in which designing the feedback control loops is a crucial issue. Since Ptolemy II is an open-source framework, designing new actors and directors or extending the existing ones is possible. A director of Ptolemy can be extended to contain the Analyze and Plan components of a MAPE-K feedback loop. The model@runtime, which is accessed by the Analyze and Plan components, consists of the actors in Ptolemy. The features provided by Ptolemy II facilitate developing self-adaptive TTCSs. These features are explained as follows.

- Although we have not done this, in principle, a Ptolemy model can interact with a real-world system through actors that mediate sensors and actuators. Such actors are called accessors because they access the physical world in a cyber-physical system [37]. Hence, the Monitor and Execute components of the MAPE-K feedback loop can smoothly be added to the model. But, a mapping from the retrieved information of the running system to the model@runtime and vice versa are needed. Since the actor model reduces the semantic gap between the models and the real-world applications, the mapping does not need much effort.

- Thanks to the Java-based definition of actors and directors in Ptolemy II, the provided MAPE-K feedback loops in Ptolemy are executable. Therefore, transforming the developed MAPE-K feedback loops to executable code is unnecessary. Moreover, a modeler is able to define more complex analysis methods and adaptation policies using facilities provided by directors and also the Java language. The directors of Ptolemy II have access to the actors and can govern their executions. Hence, upon occurrence of a change in the system and consequently updating the model@runtime, the director executes the model@runtime for each adaptation policy and selects the optimizing policy based on predicated results.

- Using composite actors in Ptolemy II, designing multiple MAPE-K feedback loops and organizing them hierarchically are possible. Each composite actor models a MAPE-K feedback loop. The coordination among different feedback loops or within a single decentralized feedback loop is accomplished through a top-level director that governs interactions among the composite actors. Furthermore, the top-level director can change the behaviors of the directors which are placed within the composite actors. In other words, the behaviors of the MAPE-K feedback loops or activities of a decentralized MAPE-K feedback loop can also be adapted.

- Due to access of directors to actors in Ptolemy, the directors are able to perform safety checks by enforcing a set of constraints on handling the messages between the actors. For instance, an actor may not be allowed to receive and process a message in some special circumstance. Therefore, deploying models that are correct by construction is possible.

Among the different MoCs in Ptolemy II, DE is best for realizing self-adaptive TTCSs because time-stamped events in DE are natural for representing the decision points in a TTCS.

4.3. A Ptolemy Template for Designing Self-Adaptive TTCSs

In this section, we explain our Ptolemy template for building self-adaptive TTCSs. This template is designed based on the coordinated actor model discussed in Section 3 and can be instantiated to build different TTCSs. In this template, each node of the traffic network is modeled by a Ptolemy actor, and the traffic controller (coordinator) is modeled as a Ptolemy director which is an extension of the DE director. In other words, the DE director realizes the scheduler part of the coordinator in the coordinated actor model and

Table 2: The mapping between elements of a TTCS, the coordinated actor model, and the proposed Ptolemy template

Entitiy in TTCS	Coordinated Actor Model	Ptolemy II
Moving Object	Message	Message
Node	Actor	Actor
Traffic Controller	Coordinator	Director

is extended by a decision-making part. Furthermore, the moving objects are modeled as messages passing through the actors. Table 2 shows how a coordinated actor model is mapped into our Ptolemy template. The actors call functions of the director to make requests or inform it about their current situations. The interactions among the director and actors of the provided template are illustrated in the sequence diagram in Fig. 10. To improve readability of Fig. 10, we have divided the operation of the director into *Decision Maker, Scheduler*, and *Channel* as three parts of the director. The interactions handled by each part are described in more detail in the following sections.

4.3.1. Ptolemy Template of a Node in a Traffic Network

The provided template for a mid-node does not depend on the type of the mid-node in different TTCSs, e.g. a station, block, or sub-track. In Ptolemy, an actor with the following template is associated with each mid-node.

- **Ports**: Each actor can have multiple input and output ports. A moving object is able to travel the route from its current node to a node adjacent to the current node in the traffic network. An output port is defined for each traveling direction towards the adjacent nodes. The entrances (exits) of an actor are determined based on the connections between its input (output) ports and the other actors.
- **Parameters**: Each actor has a set of parameters: *nodeId*, the node's identifier, and *envCond*, which shows the effect of environmental changes on the node. Based on the environmental changes, the Boolean parameter *envCond* represents availability or unavailability of the node.
- **Variables**: The capacity of each node is usually one to indicate that the node is a critical section. The information about the currently traveling moving object across the node is kept in the *inTransit* variable. The *arriveInTime* variable contains the expected time at which the current moving object will depart from the node (this time is equal to the arrival time of the current moving object at the subsequent node in its traveling route). The current moving object may need the permission of the controller to depart from the node at the time *arriveInTime*. The *transitExpire* variable keeps a time at which the current moving object issues a request to the controller to get permission for departing from the node. This time is before *arriveInTime*.
- ***attributeChanged* method**: This method is automatically called upon a change in the value of the *envCond* parameter. This method calls a suitable function of the director to inform it about the change.
- ***reject* method**: This method is purposely added to the actors to check availability of their corresponding nodes. In other words, this method determines under which conditions a moving object is rejected from entering into a node. A node is available if no moving object is passing through the node (*inTransit* is null), there is no adversarial condition in the node (*envCon* is false), and also the node has not been reserved for a moving object. The request of a moving object to enter into a node is accepted if the node is available or it has been reserved for the requesting moving object. Otherwise, the moving object is rejected from entering into the node. It is noteworthy that environmental changes in a node only affect its availability. The *reject* method calls the *checkReservation* function of the director to check whether the node has been reserved for the requesting moving object (message 10 in Fig. 10).

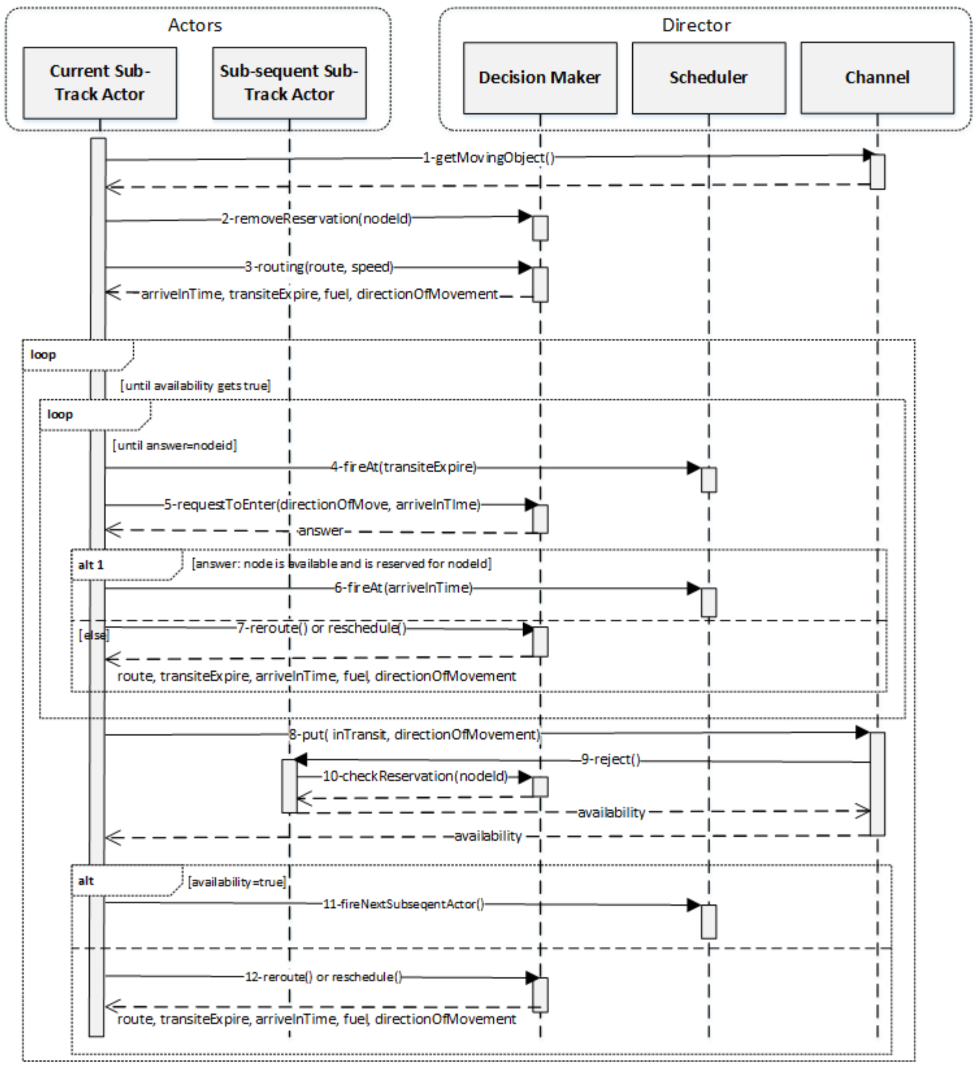

Actors
Current Sub-Track Actor
Sub-sequent Sub-Track Actor
Director
Decision Maker
Scheduler
Channel
1-getMovingObject()
2-removeReservation(nodeId)
3-routing(route, speed)
arriveInTime, transiteExpire, fuel, directionOfMovement
loop
[until availability gets true]
loop
[until answer=nodeid]
4-fireAt(transiteExpire)
5-requestToEnter(directionOfMove, arriveInTime)
answer
alt 1
[answer: node is available and is reserved for nodeId]
6-fireAt(arriveInTime)
[else]
7-reroute() or reschedule()
route, transiteExpire, arriveInTime, fuel, directionOfMovement
8-put(inTransit, directionOfMovement)
9-reject()
10-checkReservation(nodeId)
availability
availability
alt
[availability=true]
11-fireNextSubseqentActor()
12-reroute() or reschedule()
route, transiteExpire, arriveInTime, fuel, directionOfMovement

Figure 10: The interactions between sub-track actors and the director in the proposed template

- ***fire* method**: This method performs the main computation of the actor by asking the director about the traveling direction of a moving object towards the subsequent node, obtaining permission for arrival of the moving object at the subsequent node, and determining a new route or schedule for the moving object if it is necessary. In this method, by entering a moving object into a node (as shown by the message 1, the actor calls the *getMovingObject* function), *inTransit* is set to the information of the incoming moving object (the received message), and the possible reservation for the node is removed (message 2). To remove the reservation, the actor calls the *removeReservation* function of the director.

  If routing is needed, the traveling direction (output port) towards the subsequent node in the route of the moving object is determined at the moment the moving object arrives at the node (message 3). To this end, the actor calls the *routing* function of the director by passing the traveling route and speed of the current moving object. The director calculates *arriveInTime*, *transitExpire*, and the required fuel besides the exit direction. This information also is stored in the director. Then, the actor executes *fireAt(transitExpire)*, signaling the director to fire the current actor at the time *transiteExpire* (message 4). After firing the actor at the time *transitExpire*, the moving object raises a request for the controller to enter into the subsequent node in its traveling route (message 5). To this end, the actor calls the *requestToEnter* function of the director by passing the exit direction and *arriveInTime*. The director, based on its knowledge, checks availability of the subsequent node (alt 1). If the subsequent node is available, the director finds all moving objects intending to enter into the subsequent node at the time *arriveInTime* and selects one of them based on a predefined policy. This means that the subsequent node is reserved for the selected moving object. If the currently moving object is selected, it continues its traveling. It is notable that the director can reduce the speed of the current moving object by returning a new *arriveInTime* and speed, if the subsequent node will be available at a close enough future time.

In the case of selecting the current moving object to enter into the subsequent node, the actor invokes the *fireAt(arriveInTime)* method (message 6). If the director notices that the subsequent node is unavailable and has not been reserved for the current moving object, it finds a new route (if rerouting is possible), calculates new values for *arriveInTime* and *transitExpire*, and determines the required fuel (as shown by the message 7, the actor calls the *reroute* function of the director). The director is also able to reschedule all moving objects (the *reschedule* function). If the director cannot find a new route, it reduces the speed of the moving object (by calculating a new *transitExpire* and *arriveInTime*) and the moving object travels based on its previous route. In both cases, the actor executes *fireAt(transitExpire)* (message 4). After firing the actor at the time *arriveInTime*, the moving object has traveled the distance of the node and is about to leave the node. This means that the actor calls the *put* function of the director to send out a message containing the value of *inTransit* (message 8). For safety concerns, availability of the subsequent node is checked (message 9), and if it is unavailable, the moving object is prevented from entering into the subsequent node. At this time, the controller interferes (the director is called) to find a new route and calculate the new values for time variables (message 12). If the subsequent node is available, it will be fired (as shown by the message 11, the *fireNextSubsequentActor* function of the director is called).

- ***initialize* method**: The *initialize* method of the actor initializes its variables.

The above template only considers the traveling time of moving objects. However, in some cases, the residual time of a moving object in a node is greater than the traveling time. For instance, a moving object may take some time to start moving or stopping (e.g. the takeoff or landing time of an aircraft). The moving object may also need to stay for an amount of time in a node (e.g. a station in a railway system). This additional time can be set using an additional parameter of the actor or asked from the director and is added to *transitExpire* and *arriveInTime*.

The mid-node template can be slightly modified for modeling the source and destination nodes of the traffic flows. The actors corresponding to the source nodes do not have input ports. Since the source nodes are origins of the traffic, their corresponding actors do not have the *envCon* and *transiteExpire* parameters. Consequently, the actor corresponding to the source node do not also have the *attributeChanged* and *reject* methods. Instead, they have a waiting list of moving objects together with their departure times. In the *fire* method, if the source node has at least one free exit at the departure time of a moving object (capacity for departure), the moving object starts to depart. It means that *arriveInTime* of the actor corresponding to the source node is set to the arrival time of the moving object at the first mid-node in its traveling route. If the node does not have a free exit, departure of the moving object is postponed by an amount of time determined by the director. Also, the moving object may be rejected by the next node, in which case it stays in the source node for a time also determined by the director.

The actors associated with destination nodes do not have output ports, *transitExpire*, and *envCon*. Similar to the mid-node, the destination nodes are able to accept or reject a moving object based on their capacities. Therefore, they use the same *reject* method as that of the mid-node template. A moving object traveling through a destination node stops its movement at *arriveInTime*, which is set by the director.

4.3.2. Ptolemy Template of a Controller in a Traffic Network

For modeling self-adaptive TTCSs, we extend the DE director of Ptolemy II with a decision-making part. The decision maker contains several monitoring functions which are called by the actors to inform

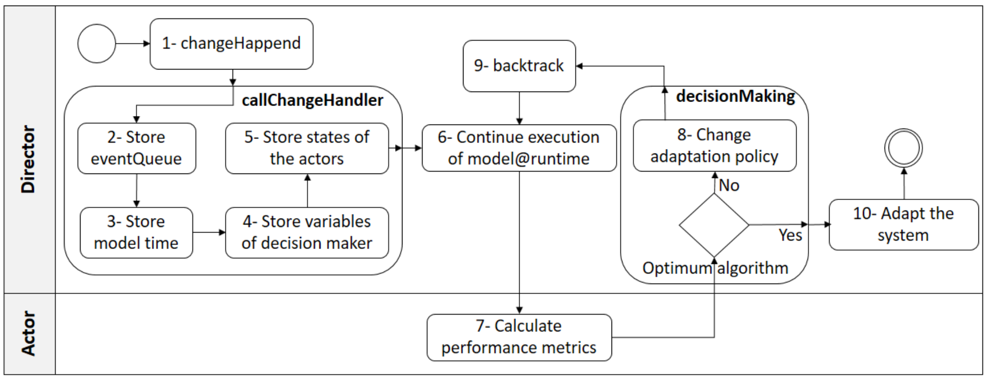

Director
Actor
1- changeHappend
callChangeHandler
2- Store eventQueue
3- Store model time
4- Store variables of decision maker
5- Store states of the actors
6- Continue execution of model@runtime
7- Calculate performance metrics
9- backtrack
decisionMaking
8- Change adaptation policy
No
Yes
Optimum algorithm
10- Adapt the system

Figure 11: The sequence of activities in the director for realizing predictive adaptation

the director about their current states. In response, the director acquires the necessary knowledge about the actors (traffic network) and returns its adaptation decision. In other words, the director employs its acquired knowledge to route, reroute, reschedule, and even to reserve a node for a moving object.

We use features of the director to check availability of the corresponding node to an actor. As mentioned in Section 4.1, when a message is sent over a channel, the *put* function of the channel is called. We override the *put* function such that it calls the *reject* function of the receiver actor. If the return value of the *reject* function indicates availability of the node corresponding to the receiver actor, the message is sent successfully. This means that the moving object is able to enter into the node. Otherwise, the moving object is rejected from entering into the node and is rerouted. Using this approach, one of the safety issues, *separation between the moving objects*, is built into the model.

4.4. Cloning the Model: A Mechanism for Predictive Adaptation

In order to implement predictive adaptation in Ptolemy II, we extend the director with three more functions; *callChangeHandler*, *decisionMaking*, and *backtrack*. The sequence of activities accomplished by the director to realize predictive adaptation is illustrated by the activity diagram in Fig. 11 and is described as follows.

- ***callChangeHandler* function**: A change in the system is reflected in the model by changing the value of the *envCond* parameter of one or several actors. The director is informed about the change when the *attributeChanged* method of an actor is called. In other words, the *attributeChanged* method calls the *changeHappened* function of the director (activity 1). To start prediction and adaptation, the director calls *callChangeHandler* to freeze the current time of the model and clone the model@runtime. To this end, the *eventQueue* of the director, the model time, the state of the decision maker, and the current states of the actors are stored (activities 2 to 5). Then, the director continues execution of the model@runtime (activity 6). During the execution, a computing actor (an actor other than the associated actors with the nodes) calculates several performance metrics such as the throughput, delay, response time, etc (activity 7). The performance metrics are measured when a moving object reaches its destination. They are calculated using the information carried by messages (moving objects). For instance, the departure time is carried by the associated message with a moving object when the moving object leaves its source node. It is possible that the moving objects are stuck in a traffic blockage (a deadlock happens) and the execution does not proceed. In this situation, the computing actor times out after passing a predefined time (e.g. after t units of time). This time is more than the expected overall execution time of the model@runtime. In other words, when the computing actor is initialized, it executes the *fireAt(t)* instruction. By firing the computing actor at the time t, it calls the *decisionMaking* function of the director and passes the calculated metrics. These activities can be performed for each adaptation mechanism (activity 8). So, the model@runtime has to be reverted back to its initial state by calling the *backtrack* function (activity 9). The clone of the model which

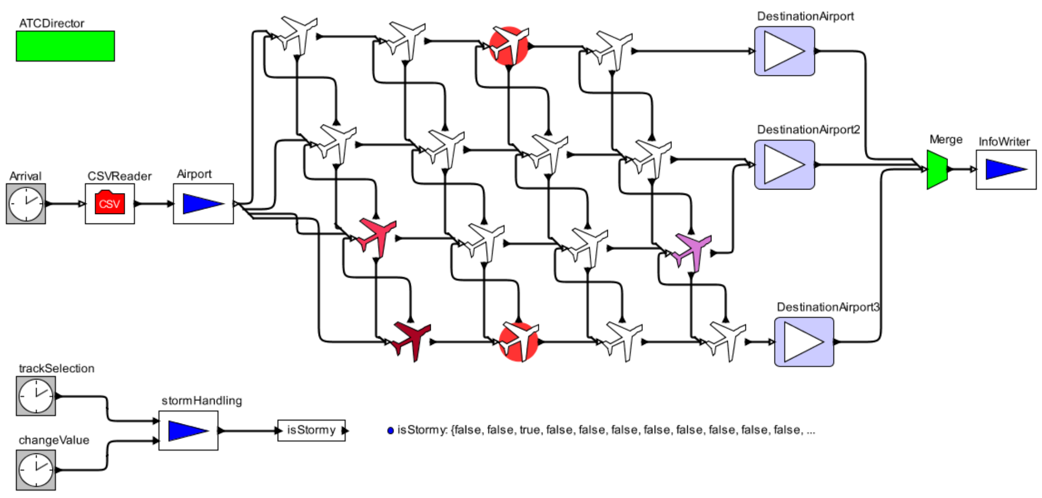


Figure 12: The Ptolemy model of an air traffic control system. The sub-tracks are illustrated with the white aircraft and the occupied ones are shown with the colorful aircraft. The red circles around the aircraft show adversarial environmental conditions in the sub-tracks.

was created at the beginning of these steps is used for this purpose. Finally, the designed adaptation is issued to the system (activity 10).

- ***decisionMaking* function**: This function receives the calculated metrics and uses a set of prediction-based adaptation policies to adapt the model. For instance, it adapts the model by switching to another rerouting or rescheduling policy. After making a decision, the *backtrack* function, described in the following item, is called.
- ***backtrack* function**: This function restores the model@runtime with the cloned one. It restores the model time, *eventQueue*, state of the decision maker, and states of the actors.

5. Case Study

We assessed usability of the coordinated actor model for realizing self-adaptive TTCSs in Section 3.4, where we described how different applications of TTCSs are mapped into the coordinated actor model. We also proposed a Ptolemy template for developing the coordinated actor model of TTCSs in Section 4.3. In this section, to have a clear judgment of how easily the coordinated actor model and also the proposed Ptolemy template are used for realizing self-adaptive TTCSs, we instantiate the proposed Ptolemy template for an ATC application and a rail traffic control system. As mentioned in Section 3.4, the main difference between these two applications is in their adaptation policies. Although, ATC and the rail traffic control system have similar traffic networks, ATC accepts more varied adaptation policies in comparison with the rail traffic control system. For instance, rerouting the trains in a metro subway system is not possible because the trains should service the passengers waiting in stations. Therefore, we illustrate effectiveness of our approach by applying it to ATC. In other words, we show how different adaptation policies affect on the behavior of an ATC example. Toward this aim, we explain different rerouting algorithms (adaptation policies) for ATC and measure performance of the system by applying each one. We also illustrate the results of applying predictive adaptation to our case study. Given a traveling plan for each moving object, our model of the system behavior is deterministic, and therefore simulation is sufficient to check for deadlock and to assess performance impacts of the adaptation policies.

5.1. ATC and the Rail Traffic Control System in the Proposed Ptolemy Template

The simplified Ptolemy model of ATC is depicted in Fig. 12. This model is built using the template provided in Section 4.3. In Fig. 12, sub-tracks of ATC are illustrated by the aircraft shapes and the

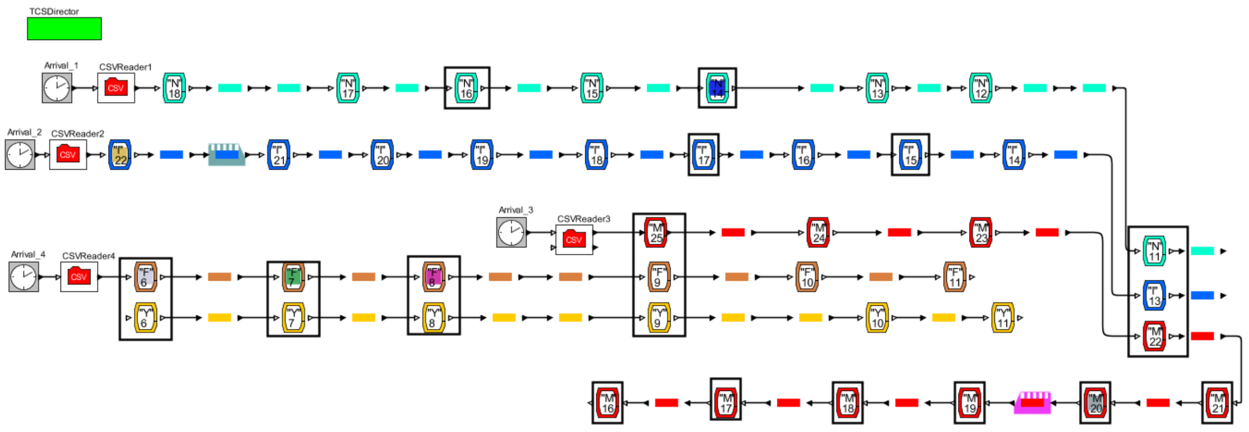


Figure 13: The Ptolemy model of a small part of the Tokyo subway system [21]. The occupied stations are shown with the colored squares. The occupied blocks are signed with the train shapes.

occupied sub-tracks are represented by a color. Each aircraft (moving object) has a unique identifier, which is shown by its color. The sub-tracks affected by adversarial environment conditions such as thunderstorms are highlighted with a red circle around them. The source and destination of the aircraft are modeled by Ptolemy actors with the captions of *Airport* and *Destination Airport*. The director of ATC model is shown as a green rectangle with the caption of *ATCDirector*. Also, the simplified Ptolemy model of a small part of Tokyo subway system, constructed based on the proposed Ptolemy template, is depicted in Fig. 13. In Fig. 13, the occupied stations are shown with the colored squares and the occupied blocks are presented with the train shapes. The green rectangle with the caption of *TCSDirector* depicts the director of the rail traffic control system model.

To compare effects of different adaptation policies on the behavior of the ATC example, we simulate arrival of the aircraft and the effect of the environmental changes on the sub-tracks. In Fig. 12, flights are initialized by the *Arrival* and *CSVReader* actors. The *Arrival* actor is a discrete clock that determines departure times of the aircraft. The *CSVReader* actor reads flight plans and other detailed information of the aircraft from a text file. The environmental changes are simulated through three actors; *TrackSelection*, *ChangeValue*, and *StormHandling*. The *TrackSelection* actor selects a sub-track (with a predetermined probability). The *ChangeValue* actor decides about how to affect on the selected sub-track, e.g. by generating or removing a thunderstorm. The *StormHandling* actor applies the made decision of *ChangeValue* to its target sub-track. For predictive adaptation, the actor with the caption of *InfoWriter* calculates and issues the values of the performance metrics to *ATCDirector* (this actor is the mentioned computing actor in Section 4.4).

Developing multiple interactive coordinated actor models is also supported by the Ptolemy framework. Fig. 14 illustrates a heterogeneous and hierarchical model of an air traffic network with four control areas and three levels of the hierarchy. In the first level of the hierarchy, two composite actors *ATC12*, and *ATC34* are coordinated by *AutonomousDirector* which is an extension of the DE director in Ptolemy II. Each composite actor in the first level of the hierarchy contains two control areas and each control area itself is modeled by a composite actor in the second level of the hierarchy. In the second level, the composite actors *ATC1* and *ATC2* (resp. *ATC3* and *ATC4*) are coordinated by *DecentralizedDirector1* (resp. *DecentralizedDirector2*) which is also an extension of the DE director. Each composite actor of the second level of the hierarchy is also a model in the third level, containing sub-track actors and an *ATCDirector*. This model can be executed at runtime. The directors of different control areas might use different rerouting algorithms and hence they have different implementations. The source airport and four destination airports are placed in the composite actors *ATC1* and *ATC4*, respectively. The flow of aircraft from *ATC1* to *ATC2* (resp. *ATC3* to *ATC4*) is managed by *DecentralizedDirector1* (resp. *DecentralizedDirector2*) which has access to both directors of the composite actors *ATC1* and *ATC2* (resp. *ATC3* and *ATC4*). *ATCDirector*s in *ATC1* and

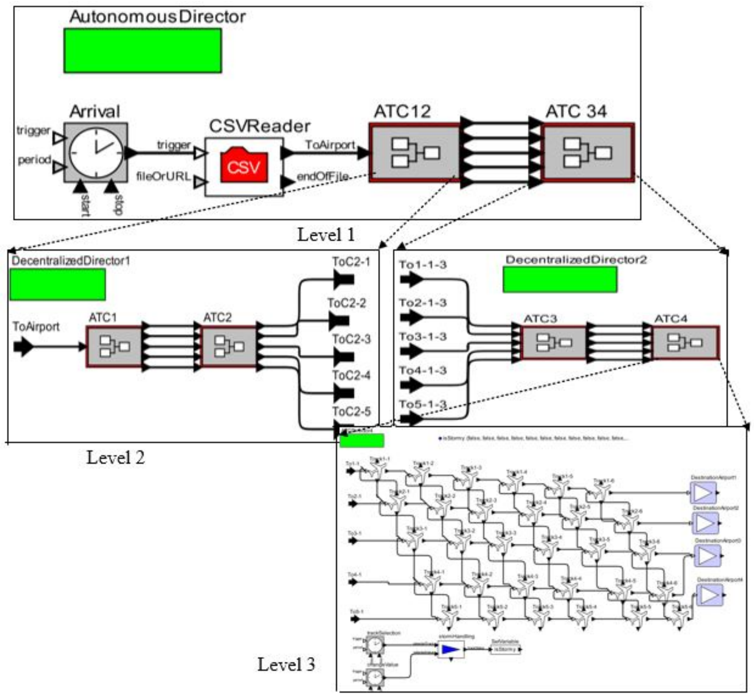


Figure 14: The hierarchical model of an air traffic network with four control areas and three levels of the hierarchy.

ATC2 calls *DecentralizedDirector1* to inform it about the current state of the traffic flow and environmental conditions in their control areas. Moreover, the flow of the aircraft from *ATC12* to *ATC34* is managed by *AutonomousDirector*.

This hierarchical model improves scalability in constructing and analyzing the model@runtime of self-adaptive TTCSs. However, compositional reasoning using the provided hierarchical model is beyond the scope of this paper. Hence, the focus of the rest of the paper is on basic building blocks of the hierarchy, i.e. a control area in the lowest level of hierarchy (level 3 in Fig. 14).

5.2. Adaptation in ATC: Dynamic Reroute Planning

An aircraft flight plan consists of a route, which is a sequence of sub-tracks traveled by the aircraft towards its destination. The controller prevents the aircraft from entering into unavailable sub-tracks in their routes by rerouting them. An unavailable sub-track is one that is affected by an adversarial environmental condition, e.g. a storm, or a sub-track that is dedicated to another aircraft. We assume that a sub-track is removed from the route when the aircraft leaves the sub-track. Suppose that the current flight route of the aircraft A is shown as the sequence $[T_1, T_2, \cdots, T_n]$. In this sequence, T_1 is the current location of the aircraft in its flight route and T_n is the destination airport. When the aircraft A wants to leave T_1 and the subsequent sub-track on its flight route (sub-track T_2) is unavailable, the coordinator applies one of the following algorithms to reroute the aircraft.

Safer Route Policy (SRP). As shown in Algorithm 1, SRP finds the feasible shortest flight route from T_1 to T_n (line 2). A feasible flight route is a flight route that does not contain an unavailable sub-track. If there is no such route, the shortest flight route with the least number of unavailable sub-tracks is returned (line 4). In Algorithm 1, the functions $FSRoute(T_1, T_n)$ and $SRoute\_LNUS(T_1, T_n)$ return the feasible shortest flight route and the shortest flight route with the least number of unavailable sub-tracks from T_1 to T_n, respectively.

Blocked Area Avoidance Policy (BAAP). Similar to SRP, BAAP tries to find the feasible shortest flight route from T_1 to T_n, which is shown in line 2 of Algorithm 2. If there is no such route, it finds the

Algorithm 1: Safer Route Policy (SRP)

```
  Input: The sub-track $T_1$ as the current position of the aircraft and the destination airport $T_n$
  Output: The new route of the aircraft
1 begin
2     route ← FSRoute($T_1$, $T_n$)
3     if route = null then
4         route ← SRoute_LNUS($T_1$, $T_n$)
5     end
7     return route
8 end
```

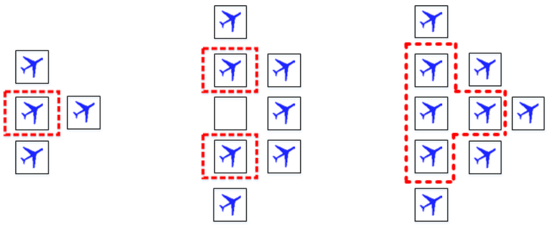


Figure 15: Three different arrangements of the aircraft in the airspace which form blocked areas. The blocked areas containing unavailable sub-tracks are shown with the dotted-rectangles. The blocked areas are avoided in the BAAP rerouting policy.

shortest route that does not pass a blocked area (line 4). The function *SRoute_WBA*(T_1, T_n) calculates the shortest route from T_1 to T_n that does not pass a blocked area. As shown in Fig 15, a blocked area is a part of the airspace in which all of its sub-tracks have been occupied and surrounded by other aircraft at the time of rerouting the aircraft A. In other words, an aircraft in a blocked area may hold its current sub-track due to the high traffic around it for a long time. Hence, the aircraft A probably would not be able to pass a sub-track in the blocked area if that sub-track is selected as a part of the aircraft's flight route. We consider three arrangements of the aircraft in the airspace which form the blocked areas. These arrangements are shown in Fig. 15. For instance, if four aircraft hold the sub-tracks such that the left shape of Fig. 15 appears in the airspace, a blocked area, shown by a dotted-rectangle, is formed. Since BAAP avoids areas with the higher traffic, it is expected to result in less flight duration compared to SRP.

Algorithm 2: Blocked Area Avoidance Policy (BAAP)

```
  Input: The sub-track $T_1$ as the current position of the aircraft and the destination airport $T_n$
  Output: The new route of the aircraft
1 begin
2     route ← FSRoute($T_1$, $T_n$)
3     if route = null then
4         route ← SRoute_WBA($T_1$, $T_n$)
5     end
7     return route
8 end
```

Shrinking Search Policy (SSP). As shown in Algorithm 3, SSP employs a multi-step searching algorithm (lines 4 to 12). At the first step, SSP tries to find a feasible shortest flight route from T_1 to T_n. If there is no such route, it tries to find a feasible shortest flight route from T_1 to T_{n-1}, and so on. If a route from T_1 to $T_i, 1 < i \leq n$, is found, the route is concatenated with the current route of the aircraft from T_i to T_n (line 7) and the resulted route is returned. Otherwise, the aircraft A holds its position in T_1

for some amount of time, and attempts to continue flying based on its current flight route (*FRoute* as the current flight route of the aircraft is returned in line 14). If $FRoute = [T_1, \cdots, T_i, T_{i+1}, \cdots, T_n]$, the functions *Head*(*FRoute*), *Tail*(*FRoute*), and $Ahead(T_{i+1})$ return T_1, T_n, and T_i, respectively. Furthermore, the function *Subseq*(*FRoute*, T_i) returns $[T_i, \cdots, T_n]$ and $Concat(R_1, R_2)$ returns the route resulting from concatenation of the routes R_1 and R_2. Since SSP attempts to find the feasible shortest flight route for the largest possible part of the flight route, it is expected to result in the less blocking probability for the aircraft in the airspace.

Algorithm 3: Shrinking Search Policy (SSP)

```
Input: The route FRoute as the current flight route of the aircraft
Output: The new route of the aircraft
1  begin
2      $T_1 \leftarrow Head(FRoute)$
3      $T \leftarrow Tail(FRoute)$
4      while $T_1 != T$ do
5          $route \leftarrow FSRoute(T_1, T)$
6          if $route != null$ then
7              $route \leftarrow Concat(route, Subseq(FRoute, T))$
9              return route
10         end
11         $T \leftarrow Ahead(T)$
12     end
14     return FRoute
15 end
```

Combined Rerouting Policy (CRP). As shown in Algorithm 4, CRP uses a combination of SRP, BAAP, and SSP to reroute the aircraft (lines 6 to 20). In other words, based on the current configuration of the model (airspace), which is observed by the analyzer, the planner chooses the most suitable algorithm for rerouting the aircraft. The current configuration of the airspace is returned by the function *FindAConfig()* (line 2). This function returns 0 if the airspace has a normal situation, 1 if a blocked area is detected in the airspace, and 2 if a global congestion happens in the airspace. Global congestion happens when the number of rejected arrivals at the sub-tracks and the number of unavailable sub-tracks pass the thresholds Th_r, and Th_u, respectively. Suppose that the airspace has a predetermined load of the aircraft (line 5), unless a global congestion is detected. In CRP, the default rerouting algorithm (line 8) is SRP, but upon detecting a blocked area in the airspace, the planner switches to BAAP to reduce the flight duration of the aircraft (line 12). Moreover, the planner will switch to SSP when global congestion is detected (line 16). Also, it decreases the airspace load by restricting the number of takeoffs from the source airports (line 17). Then by arrival of a percentage of the aircraft (which is shown by P_A in the experiments) at their destinations, the aircraft delayed in their source airports are permitted to takeoff. The function *LoadUp()* in Algorithm 4 returns the load of the airspace to its predetermined value if the load has been decreased and a percentage of the aircraft have arrived at their destinations. The function *UnLoad()* decreases the load if, in the presence of the global congestion, the load has not changed.

Predictive adaptation using Runtime Performance Prediction (RPP mechanism). When a storm happens to the environment of an ATC, the coordinator uses one of the described algorithms to reroute the aircraft. This way, the coordinator prevents the aircraft from traveling through the sub-tracks affected by the storm. However, the coordinator can improve its adaptation mechanism by anticipating the future behavior of the system. For instance, the coordinator looks over the future behavior of the system to predict whether by employing the current rerouting algorithm for adapting the system, a system requirement will be violated. Upon detecting a violation, it adapts the system by switching to another rerouting algorithm. This is the second level of the adaptation mentioned in Section 2.1. In our implementation of RPP, the coordinator

Algorithm 4: Combined Rerouting Policy (CRP)

```
Input: The route $FRoute$ as the current flight route of the aircraft
Output: The new route of the aircraft
1  begin
2      $c \leftarrow FindAConfig()$
3      $T_1 \leftarrow Head(FRoute)$
4      $T_n \leftarrow Tail(FRoute)$
5      $LoadUp()$
6      switch $c$ do
7          case 0 do
8              $route \leftarrow SRP(T_1, T_n)$
9              break
10         end
11         case 1 do
12             $route \leftarrow BAAP(T_1, T_n)$
13             break
14         end
15         case 2 do
16             $route \leftarrow SSP(FRoute)$
17             $UnLoad()$
18             break
19         end
20     end
22     return $route$
23 end
```

switches between SSP and BAAP. Based on the experimental results in Section 5.3, we will explain under which conditions the coordinator switches between SSP and BAAP.

5.3. Performance Evaluation of Dynamic Rerouting Policies

The performance metrics for comparing the described rerouting algorithms in Section 5.2 include the number of aircraft that have arrived at their destinations, the number of aircraft stuck in a deadlock in the airspace, the average delay of the aircraft in their source airports, the number of aircraft blocked in their source airports, and the average flight duration of the aircraft. It is notable that these performance metrics are measured when an aircraft arrives at its destination (we remind the reader that the *InfoWriter* actor in Fig. 12 calculates the performance metrics, and each message carries the required time information such as the planned departure time of the aircraft from its source airport, its real departure time, etc.). A deadlock is detected when none of the aircraft in the airspace change their current sub-tracks within a pre-determined time. This situation happens when the controller is unable to reroute the aircraft toward their destinations and they are stuck in a traffic blockage. In addition to the deadlock, the separation between the aircraft and their safe rerouting in a stormy weather are built into the model. A sub-track is a critical section, denoting the safe distance between two aircraft. The controller never triggers a sub-track actor to process a new message (aircraft) while the sub-track actor is marked as unavailable (it is occupied or stormy).

In our experiments, we assume one unit of time for takeoff/landing and crossing an aircraft along a sub-track. Also, *transitExpire* is 0.2 unit of time before *arriveInTime*. If an aircraft cannot find a new route to its destination, it takes its current position for one more unit of time. Furthermore, the capacity of each source airport, destination airport, and sub-track is one. We model an ATC example with two source airports, eight destination airports, and thirty sub-tracks. To see the differences among different rerouting policies, we intentionally increase crossovers of aircraft; fifty aircraft plan to fly from a north airport to one of the three south destination airports and the same number of aircraft intend to fly from the west airport to the five destination airports in the east. In this setting, deadlock is inevitable. For each source airport, twenty batches of flight plans are generated, where each flight plan contains the initial route of the aircraft and the departure time from its source airport. The initial route of an aircraft is the shortest route from its source airport to its destination. The departure times of the aircraft, and the times of the weather changes are produced using Poisson processes with parameters λ in $\{0.5, 2, 3.5\}$ (the mean interval time between two events) and 1 as μ, respectively [1]. Per each adaptation policy and each value of λ, the model is executed 150 times for one batch of flight plans and average of the performance metrics are calculated.

The effect of executing SRP, BAAP, SSP, CRP, and RPP on the performance metrics are shown in Figures 16 to 20. In higher values of λ, the traffic is lighter. Hence, for a constant value of departing aircraft (100), increasing the value of λ results in increasing the number of arrived aircraft at their destinations, decreasing the number of blocked aircraft in the source airports, decreasing delays in the source airports, and decreasing the flight duration. Although, by increasing λ the number of aircraft blocked in the airspace decreases, the change of its values for $\lambda = 0.5$ and 2 is small.

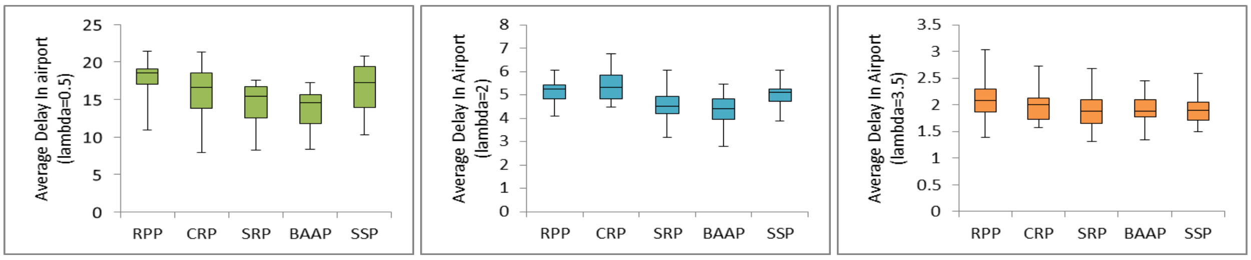


Figure 16: The average delay of the aircraft in their source airports for different values of λ, and $\mu = 1$.

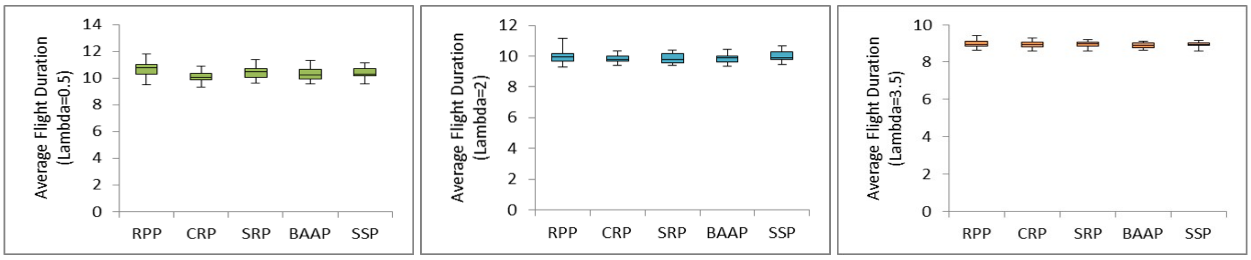


Figure 17: The average flight duration of the aircraft for different values of λ, and $\mu = 1$.

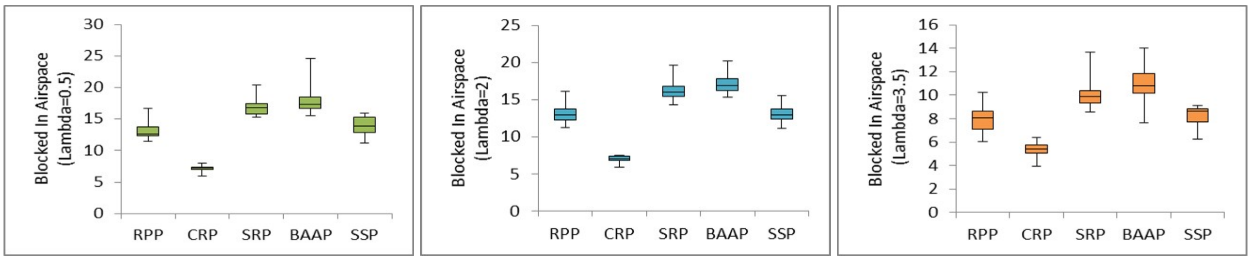


Figure 18: The number of aircraft which are stuck in a deadlock in airspace for different values of λ, and $\mu = 1$.

The BAAP rerouting policy performs better than SSP and SRP policies based on the metrics of average delay in the source airports and the average flight duration. The BAAP policy reduces the average flight

[1]The source code which is used for these experiments is uploaded in `http://rebeca.cs.ru.is/files/ATC.zip`

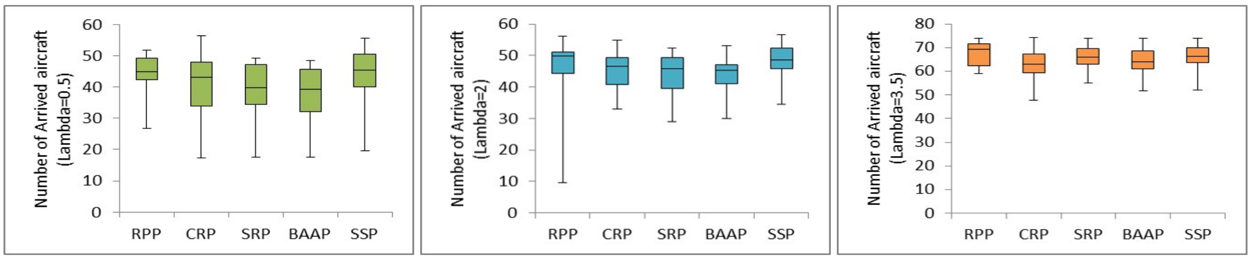


Figure 19: The number of arrived aircraft at their destination airports for different values of λ, and $\mu = 1$.

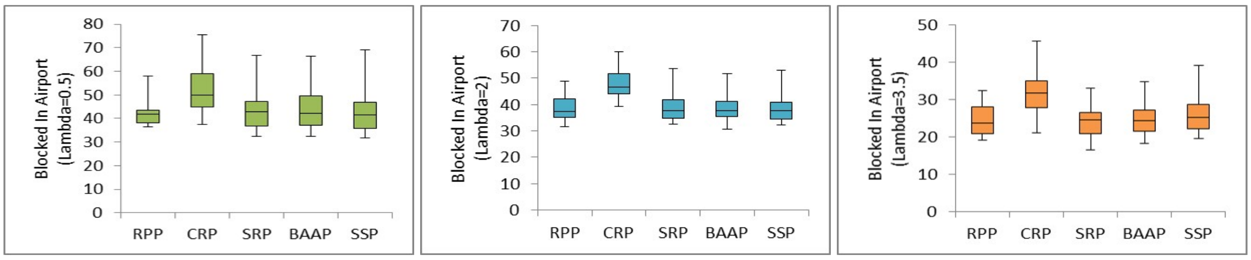


Figure 20: The number of aircraft blocked in their source airports for different values of λ, and $\mu = 1$.

duration because if it cannot find a feasible shortest route for a rejected aircraft, it tries to find a route avoiding passing the aircraft from the blocked areas. When the aircraft are not allowed to pass through the blocked areas, they rarely need to change their route and also hold the sub-tracks for less time. Consequently, the average delay of the aircraft in the source airport decreases. However, passing the aircraft from the blocked areas in light traffic does not have a same effect as their crossing through the blocked areas in heavy traffic. Therefore, conservation of BAAP for rerouting the aircraft in light traffic might increase the average delay of the aircraft in their source airports. By increasing the number of blocked areas in the airspace, BAAP is not successful in rerouting the aircraft, and consequently the number of blocked aircraft in the airspace increases.

SSP works better than BAAP and SRP in terms of the number of aircraft stuck in a deadlock, the number of aircraft arrived at their destinations, and consequently the number of blocked aircraft in the source airports. The cause is that SSP attempts to find the feasible shortest flight route for the longest part of the flight route. However, as SSP is a conservative policy in comparison with the other policies in finding a route, it increases the average delay in the source airport. The reason is that it holds an aircraft in its source airports until it finds new routes for the rejected aircraft holding their positions in the sub-tracks. For the same reason, SSP performs worse than BAAP and SRP in terms of the number of blocked aircraft in the source airports for large values of λ.

In our implementation of CRP, $T_r = 20$, $T_u = 10$, and $P_A = 50\%$. As we expected, CRP decreases the average flight duration and the number of aircraft stuck in a deadlock.

In implementation of RPP, the controller is adapted by switching between SSP and BAAP. SSP is the default rerouting policy of the controller and performs better than BAAP in terms of the number of the aircraft stuck in the airspace. But it has a higher average flight duration compared to BAAP. If a storm happens, the runtime performance prediction method is applied; if using SSP the predicted number of the aircraft that are stuck in the airspace exceeds 18 and its average delay in the airport exceeds 13.61, the planner switches its routing policy to BAAP. The intuition behind this change is that in this situation SSP does not perform better than BAAP even in terms of the number of aircraft stuck in the airspace. Furthermore, if the controller reroutes the aircraft using the BAAP rerouting policy, the predicted number of the blocked aircraft in a deadlock becomes greater than 13.86, and the average delay in an airport becomes greater than 16.49, the controller switches to SSP. As a result, the number of the aircraft that are blocked in the airspace decreases. This means that without having a considerable increase in the average flight duration, most of the aircraft can find a route toward their destinations.

6. Related Work

In this section, we investigate the most recent state of the art approaches for developing self-adaptive systems. To classify related work, we benefit from the taxonomy provided in [29] where research in the self-adaptive community is categorized based on the important characteristics of self-adaptation. These characteristics influence the realization of self-adaptive systems. We focus on an *adaptation control* aspect, among other aspects, which is related to how the adaptation engine of a self-adaptive system is constructed. Hence, we talk about the Feedback Control Loop (FCL) and the solutions inspired by FCL. Regarding software engineering aspects of self-adaptive systems, there are two most prominent approaches, whose architectures are mapped with FCL, for realizing self-adaptive systems: the MAPE-K feedback loop and the Rainbow framework. However, to prove the correctness of a self-adaptive system, the effects of FCL on the system behavior should be validated [26]. Therefore, we present the application of formal verification in analyzing self-adaptive systems in a separate category. We also study several approaches taking inspiration from the model predictive control [54] in a separate category. The approaches presented in this category use models to predict the future system behavior and reason about the outcome of the adaptation.

6.1. Adaptation Control

Feedback control loop. Autonomic systems usually form a generic FCL consisting of elements for collecting information from various resources, analyzing the information, deciding about evolving situations, and informing users or acting on the system [38]. In [38], this generic FCL is called Autonomic Feedback Loop. FCL is also a central element of control theory [26] that provides mathematically-based adaptation strategies for developing self-adaptive systems [39]. However, FCL is a key aspect of engineering self-adaptive systems [40]. The approach proposed in [40] introduces an actor-oriented modeling language, called Feedback Control Definition Language (FCDL), for modeling adaptable FCLs. In this approach, the components of a FCL, which are sensors, actuators, filters, and decision makers, are represented by the actors. Different FCLs can be organized hierarchically or can coordinate with each other. This work is completed in [41] by developing the ACTRESS toolkit. ACTRESS uses a set of model-to-model transformations to execute FCLs and the SPIN model checker to verify their connectivity and reachability properties. In comparison with our approach, the adaptation based on predication is not supported in the ACTRESS toolkit.

Rainbow framework. The Rainbow framework [42] is one of the well known approaches for realizing architecture-based self-adaptive systems. Rainbow consists of three layers: system layer, architecture layer, and translation layer. The probes and effectors, and all interfaces that are needed to access the managed resources are defined in the system layer. The architecture layer consists of *Model manager* that provides access to the architectural model of the system, *Constraint evaluator* that detects problems by evaluating the architectural model, *Adaptation engine* that triggers adaptations, and *Adaptation executor* that performs adaptation actions. The translation layer bridges the semantic gap between two mentioned layers. This framework is inherently centralized and, unlike our approach, does not support hierarchical modeling. Stitch language [43, 44], which is based on the utility theory, is used to define adaptation strategies in Adaptation engine of Rainbow. Therefore, the best adaptation strategy that makes an appropriate trade-off between the competitive objectives is selected by comparing utilities resulted from executing alternative adaptation strategies. However, the strategy selection mechanism in Stitch may leave the system in an undesirable state. The work [45] proposes a language to model impacts of the adaptation tactics on objectives of a system. This language uses a set of probabilistic expressions to model both uncertainty in the outcome of each adaptation tactic and evolution of the environment during the adaptation. Hence, formal semantics of the language is Discrete Time Markov Chain (DTMC). The authors show how the proposed impact model can be used in the context of the Rainbow framework. Since each adaptation strategy in Rainbow is a collection of adaptation tactics, its impact is calculated considering the impacts of the tactics. Finally at runtime, an adaptation strategy with the maximum utility is selected, where the utility is a function of the impact value. In comparison with our approach, the proposed approach in [45] uses probabilistic models. It also synthesizes the adaptation strategy with the maximum utility at runtime. In our approach, to find an adaptation policy maximizing an objective of the system, the model should be executed for each adaptation policy.

MAPE-K feedback loop. In the IBM's MAPE-K reference model [3], each one of the software resources is managed through an autonomous manager that consists of Monitor, Analyze, Plan, Execute, and the Knowledge component. Referring to [46], the autonomous managers do not provide an explicit support for interacting in a decentralized architecture. Therefore, the authors in [46] proposed a reference model, in which the self-adaptive units use coordination models to access the data required for interaction. A self-adaptive unit manages a part of the managed resources. This model is extended in FORM [9] that is a formally specified reference model to describe the architectural characteristics of the distributed self-adaptive systems. The main focus of this approach is on the decentralized control in a distributed system and, unlike our approach, it does not provide an implementation platform. From the same viewpoint, a set of patterns for modeling coordination among the MAPE-K activities in a decentralized self-adaptive system is introduced in [32].

Vogel et.al. [47] presented EUREMA, a model-driven engineering approach to specify and execute multiple MAPE-K feedback loops. EUREMA modeling language is extended to Deurema [4] that describes collaboration aspects in an adaptive system of systems. To realize the collaboration, the knowledge is exchanged between the coordinating MAPE-K feedback loops. However, to execute the MAPE-K feedback loops, they are interpreted at runtime and interpretation impacts on efficiency of the approach. Using our proposed implementation platform, no effort for transforming models into executable code is needed, since the models in Ptolemy are executable. Furthermore, EUREMA does not support predicting a property violation at runtime.

6.2. Verification at Runtime and Design time

PobSAM [7] is an actor-based approach to model self-adaptive systems. It consists of a set of actors, views, and autonomous managers. The managers govern and adapt the behaviors of the actors using policies. HPobSAM [48] is an extension of PobSAM for hierarchical modeling of self-adaptive systems. In [6], Rebeca [49] is used to check LTL properties over PobSAM models at design time. Unlike our approach, PobSAM (HPobSAM) is used for modeling and analyzing self-adaptive systems at design time. A number of approaches such as [13, 14, 15] use state-based models for runtime verification of self-adaptive systems. Forejt et al. [13] proposed incremental runtime verification of Markov Decision Processes (MDPs) that are described in the PRISM language. The MDP of a PRISM model is constructed incrementally by inferring a set of states needed to be rebuilt. The constructed MDP is verified using an incremental verification technique. A parametric approach for runtime verification of Discrete Time Markov Chains (DTMCs) is presented in [14, 15]. In this method, probabilities of the transitions are given as variables that get values at runtime. None of the mentioned approaches model the MAPE-K feedback loop explicitly. In comparison with the state-based models, our coordinated actor model is in a higher level of abstraction and consequently is usable and easy to build.

ActiveFORM [12] uses timed automata to model each component of the MAPE-K feedback loop, and TCTL to specify properties of the adaptation behavior. To realize self-adaptation, provided models are transformed to code which is executed on a virtual machine. Although ActiveFORM provides a tool for verifying adaptation goals at runtime as well as design time, it does not predict whether properties are violated at runtime. Unlike ActiveForm, our approach is scalable, since it supports developing multiple interactive MAPE-K feedback loops. Modeling decentralized interactive MAPE-K feedback loops at design time using Multiagent Abstract State Machine (ASM) is carried out in [50, 51]. In this approach, each adaptation concern is modeled by a MAPE-K feedback loop. The computation of each feedback loop can be distributed on a set of agents and the behavior of each MAPE-K component is specified by ASM transition rules. To detect interference between different feedback loops and assure correct behaviors of the MAPE-K feedback loops, ASM models are verified using the AsmetaSVM tool at design time. The work of [51] is an extended version of [50], in which a framework for modeling and analyzing distributed self-adaptive systems is presented, and an operational semantics for the pattern describing interactions among multiple MAPE-K feedback loops is given. In addition to verification using AsmetaSVM, the correctness of the MAPE-K feedback loops is evaluated by simulating different adaptation scenarios. However, the focus of this work is on verifying the MAPE-K feedback loops or validating their behaviors at design time. Furthermore, unlike our approach, it does not consider modeling timing aspects and quality properties.

6.3. Predictive Adaptation

In [52], an analysis technique based on model checking of Stochastic Multi-player Games (SMGs) is proposed to compare benefits of employing different proactive adaptation algorithms at design time. In contrast to our approach in which an adaptation process is run in reaction to an environmental change, the proactive adaptation mechanisms adapt the system before a demand for the adaptation has arisen. In [52], the system and the environment are two players of a SMG, while the system attempts to select the adaptation tactics that maximize its utility function. The approach also considers the latency of each tactic that is defined as the time it takes for the adaptation tactic to cause an influence on the system (a latency-aware approach). This approach uses a model to reason about the outcome of the adaptation tactics. In comparison with this approach, our adaptation mechanism is not latency-aware. The work in [52] is used at design time and does not model the MAPE-K feedback loop. Three approaches for latency-aware proactive self-adaptation that are used at runtime and model the MAPE-K feedback loop are PLA [16], CobRa [53], and QoSMOS [11]. These approaches use a model of the system to predict the future system behavior. The differences between PLA and CobRa are studied in [54], where the main differences are summarized by three items: predication that states whether the approach predicts the behavior of the system and the behavior of the environment, goal model that shows how properties of the system are represented, and actuation that explains how the designed adaptation is executed.

PLA discretizes the execution time of the system in decision periods and starts a decision process at the start of each period. In a decision process, a probabilistic model checker proactively selects an adaptation strategy that maximizes the utility of the system over a look-ahead horizon. The model checker synthesizes the best adaptation strategy by resolving the non-deterministic choice among a set of adaptation strategies. This approach is realized through the composition of the system model, the stochastic model of the environment, and the adaptation tactics. The goals in PLA are expressed through a utility function that is optimized. The designed adaptation in PLA is executed via a set of tactics that can modify parameters of the system or lead to the system-wide changes. However, the models in PLA are expressed using the PRISM language that is a state-based language and increases the semantic gap between the model and its corresponding real-world application. Similar to PLA, our approach can predict the system behavior and can include a prediction of the environment behavior. But our approach does not use a probabilistic model of the system. While our approach finds the best adaptation strategy by executing the model@runtime several times, PLA synthesizes the adaptation strategy. Unlike PLA, in which the adaptation decision phase runs periodically at a fixed interval, we run an adaptation decision phase whenever a change occurs in the environment. Moreover, nonfunctional requirements in our approach are explained in terms of the performance metrics, which should be calculated and be optimized. The adaptation in our approach is limited to modifying parameters of the system or switching between different rerouting/rescheduling algorithms. It is notable that we have not implemented the Monitor and Execute components of the MAPE-K feedback loop.

CobRa [53] as a control-based approach on proactive self-adaptation enables a system to achieve its multiple conflicting goals through controlling multiple parameters. CobRa models the MAPE-K feedback loop in which the Knowledge component contains a dynamic mathematical model of the system. This model formulates the relationship between the control parameters and a set of indicators that represent how much the goals of the system are satisfied. The system model can predict the future behavior of the system and be improved dynamically through learning. Based on the system model, CobRa solves an optimization problem to minimize a cost function over a look-ahead horizon. The answer to this optimization problem is a sequence of adaptation actions. Like PLA, CobRa discretizes the execution time of the system in decision periods and obtains the sequence of the adaptation actions at the start of each decision period. Unlike PLA, CobRa does not include a prediction of the environment behavior. The adaptation in CobRa is limited to changing control parameters of the system. CobRa attempts to keep the values of the indicators around the given reference values as the input.

QoSMOS is proposed in [11] for developing adaptive service-based systems. It integrates a set of pre-existing tools to model the MAPE-K feedback loop. To optimize the quality of service requirements, QoSMOS adapts the system based on the results obtained from predicting the future behavior of the system. It selects a policy that results in the best quality of the service. In this approach, the model@runtime is expressed by the PRISM language. Unlike our approach, QoSMOS does not support developing multiple interactive MAPE-K feedback loops. The coordinated actor model is at a higher level of the abstraction compared to the state-based model of PRISM. Although it is not explicitly stated in QoSMOS, unlike our approach, QoSMOS is a latency-aware approach on self-adaptation. The requirements in QoSMOS are explained through Computation Tree Logic (CTL), Probabilistic CTL (PCTL), Continuous Stochastic Logic (CSL), etc. Representing the predicted environment behavior is not a concern in QoSMOS. The models in QoSMOS are parameterized with configurable parameters and the values of the parameters are selected in a way that the goals are satisfied.

7. Conclusions and Future Work

In this paper, we proposed an approach to realize large-scale self-adaptive TTCSs through a set of MAPE-K feedback loops. We consider a coordinated actor model for each MAPE-K feedback loop and use multiple interactive coordinated actor models to design a large-scale self-adaptive TTCS. A coordinator includes the Analyze and Plan activities of the MAPE-K feedback loop and has access to an actor-based model@runtime maintained for safety check and performance analysis. We also provided a general mapping between the coordinated actor model and different applications of self-adaptive TTCSs. To provide support for designing and analyzing self-adaptive TTCSs, we proposed a template based on the coordinated actor model in Ptolemy II, where the DE director of Ptolemy is extended to model the coordinator. The support for hierarchical modeling in Ptolemy II enables us to develop multiple interactive coordinated actor models to perform decentralized adaptations. We discussed the benefits of using Ptolemy II as a framework for developing large-scale self-adaptive systems. We instantiated the proposed template to develop traffic control systems for two different applications. To study the applicability of the coordinated actor model and consequently our Ptolemy template in building self-adaptive TTCSs, we modeled a case study in the domain of ATC and illustrated its behavior under different adaptation policies. The main benefits of the proposed approach in this paper are summarized as follows.

1. *Scalability*: Decentralization is one of the main characteristics of self-adaptive systems where a centralized MAPE-K feedback loop is not sufficient for designing and adapting large-scale systems. Using multiple interactive coordinated actor models enhances scalability in design and analysis of large-scale self-adaptive TTCSs. Furthermore, the support for hierarchical modeling in the proposed implementation platform enables us to develop and analyze large-scale self-adaptive systems based on multiple interactive coordinated actor models.
2. *Reliability*: The coordinated actor model supports correct by construction design. In other words, several safety concerns such as separation between the moving objects or safe navigation in adversarial environmental conditions are built into the model. Moreover, the coordinated actor model supports predictive adaptation to detect and prevent possible future failures. These features are implemented in our proposed template, since the template maintains an *executable* model@runtime in the Knowledge component of an *executable MAPE-K feedback loop.*
3. *Usability*: We believe that the coordinated actor model and our implementation platform promise usability of the approach due to the following reasons. First, the coordinated actor model is able to capture the characteristics of our problem domain applications which *reduces the semantic gap* between the model and the real-world applications, and it is easy to understand and be used to build a model. In other words, the coordinated actor model is aligned with the structure of TTCSs. Second, the proposed implementation platform supports *visualizing the architecture* that aids modelers in building accurate models. However, evaluating the usability of the approach needs further study that has not been performed in this paper.

As a future work, we will exploit features of the proposed implementation platform to decompose a large-scale TTCSs into smaller control areas and enrich its analysis facilities with a verification engine to support compositional verification of large-scale self-adaptive TTCSs against complicated safety and liveness properties.

Acknowledgment

The work on this paper has been supported in part by the project "Self-Adaptive Actors: SEADA" (nr. 163205-051) of the Icelandic Research Fund.

References

[1] C. Ptolemaeus, System Design, Modeling, and Simulation: Using Ptolemy II, Ptolemy. org Berkeley, CA, USA, 2014.

[2] M. Bagheri, I. Akkaya, E. Khamespanah, N. Khakpour, M. Sirjani, A. Movaghar, E. A. Lee, Coordinated actors for reliable self-adaptive systems, in: O. Kouchnarenko, R. Khosravi (Eds.), Formal Aspects of Component Software: 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers, Springer International Publishing, Cham, 2017, pp. 241–259.

[3] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[4] S. Wätzoldt, H. Giese, Modeling collaborations in adaptive systems of systems, in: Proceedings of the 2015 European Conference on Software Architecture Workshops, Dubrovnik/Cavtat, Croatia, September 7-11, 2015, 2015, pp. 3:1–3:8.

[5] B. H. C. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, N. M. Villegas, Using models at runtime to address assurance for self-adaptive systems, Springer International Publishing, 2014, pp. 101–136.

[6] N. Khakpour, S. Jalili, C. Talcott, M. Sirjani, M. Mousavi, Pobsam: Policy-based managing of actors in self-adaptive systems, Electronic Notes in Theoretical Computer Science 263 (2010) 129 – 143, proceedings of the 6th International Workshop on Formal Aspects of Component Software (FACS 2009).

[7] N. Khakpour, S. Jalili, C. L. Talcott, M. Sirjani, M. R. Mousavi, Formal modeling of evolving self-adaptive systems, Science of Computer Programming 78 (1) (2012) 3–26.

[8] C. Ghezzi, A. Molzam Sharifloo, Dealing with non-functional requirements for adaptive systems via dynamic software product-lines, in: R. de Lemos, H. Giese, H. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems II, Vol. 7475 of LNCS, Springer Berlin Heidelberg, 2013, pp. 191–213.

[9] D. Weyns, S. Malek, J. Andersson, Forms: Unifying reference model for formal specification of distributed self-adaptive systems, ACM Trans. Auton. Adapt. Syst. 7 (1) (2012) 8:1–8:61.

[10] J. Zhang, H. J. Goldsby, B. H. Cheng, Modular verification of dynamically adaptive systems, in: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD '09, ACM, New York, NY, USA, 2009, pp. 161–172.

[11] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, G. Tamburrelli, Dynamic qos management and optimization in service-based systems, Software Engineering, IEEE Transactions on 37 (3) (2011) 387–409.

[12] M. U. Iftikhar, 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, SEAMS 2014, ACM, New York, NY, USA, 2014, pp. 125–134.

[13] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, M. Ujma, Incremental runtime verification of probabilistic systems, in: S. Qadeer, S. Tasiran (Eds.), Runtime Verification, Vol. 7687 of LNCS, Springer Berlin Heidelberg, 2013, pp. 314–319.

[14] A. Filieri, G. Tamburrelli, Probabilistic verification at runtime for self-adaptive systems, in: J. Cámara, R. de Lemos, C. Ghezzi, A. Lopes (Eds.), Assurances for Self-Adaptive Systems, Vol. 7740 of LNCS, Springer Berlin Heidelberg, 2013, pp. 30–59.

[15] A. Filieri, C. Ghezzi, G. Tamburrelli, Run-time efficient probabilistic model checking, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 341–350.

[16] G. A. Moreno, J. Cámara, D. Garlan, B. R. Schmerl, Proactive self-adaptation under uncertainty: a probabilistic model checking approach, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, 2015, pp. 1–12.

[17] International Civil Aviation Organization (ICAO), North atlantic operations and airspace manual (2016).

[18] Airport and air traffic control system, available at `https://www.princeton.edu/~ota/disk3/1982/8202/8202.PDF` (1982).

[19] D. Bertsimas, G. Lulli, A. Odoni, The air traffic flow management problem: An integer optimization approach, in: A. Lodi, A. Panconesi, G. Rinaldi (Eds.), Integer Programming and Combinatorial Optimization, Vol. 5035 of LNCS, Springer Berlin Heidelberg, 2008, pp. 34–46.

[20] W. Fang, S. Yang, X. Yao, A survey on problem models and solution approaches to rescheduling in railway networks, IEEE Transactions on Intelligent Transportation Systems 16 (6) (2015) 2997–3016.

[21] Subway map, available at `http://www.haneda-tokyo-access.com/en/transport/line.html`.

[22] Flying the atlantic ocean, available at `http://www.bcavirtual.com/VA%20flight%20School/atlanticflyiingrules.htm`.

[23] Z. Sharifi, M. Mosaffa, S. Mohammadi, M. Sirjani, Functional and performance analysis of network-on-chips using actor-based modeling and formal verification, ECEASST 66.
URL http://journal.ub.tu-berlin.de/eceasst/article/view/890

[24] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, J. Whittle, Software Engineering for Self-Adaptive Systems: A Research Roadmap, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 1–26.

[25] D. Weyns, Software engineering of self-adaptive systems: an organised tour and future challenges.

[26] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, M. Shaw, Engineering Self-Adaptive Systems through Feedback Loops, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 48–70.

[27] R. de Lemos, H. Giese, H. Müller, M. Shaw, J. Andersson, et al., Software engineering for self-adaptive systems: A second research roadmap, in: R. de Lemos, H. Giese, H. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems II, Vol. 7475 of LNCS, Springer Berlin Heidelberg, 2013, pp. 1–32.

[28] F. D. Macas-Escriv, R. Haber, R. del Toro, V. Hernandez, Self-adaptive systems: A survey of current approaches, research challenges and applications, Expert Systems with Applications 40 (18) (2013) 7267 – 7279.

[29] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, C. Becker, A survey on engineering approaches for self-adaptive systems, Pervasive and Mobile Computing 17 (Part B) (2015) 184 – 206, 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian.

[30] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, 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, ACM, New York, NY, USA, 2012, pp. 67–79.

[31] R. Calinescu, C. Ghezzi, M. Kwiatkowska, R. Mirandola, Self-adaptive software needs quantitative verification at runtime, Commun. ACM 55 (9) (2012) 69–77.

[32] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, K. M. Göschka, On Patterns for Decentralized Control in Self-Adaptive Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 76–107.

[33] P. Inverardi, P. Pelliccione, M. Tivoli, Towards an assume-guarantee theory for adaptable systems, in: Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on, 2009, pp. 106–115.

[34] A. David, K. G. Larsen, A. Legay, M. H. Møller, U. Nyman, A. P. Ravn, A. Skou, A. Wasowski, Compositional verification of real-time systems using ecdar, International Journal on Software Tools for Technology Transfer 14 (6) (2012) 703–720.

[35] S. X. Yang, Y. Hu, M. Q. h. Meng, A knowledge based ga for path planning of multiple mobile robots in dynamic environments, in: 2006 IEEE Conference on Robotics, Automation and Mechatronics, 2006, pp. 1–6.

[36] L. E. Parker, Multiple Mobile Robot Teams, Path Planning and Motion Coordination in, Springer New York, New York, NY, 2009, pp. 5783–5800.

[37] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvellet, B. Osyk, M. Weber, A component architecture for the internet of things, to Appear in Proceedings of the IEEE (2018).

[38] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, F. Zambonelli, A survey of autonomic communications, ACM Trans. Auton. Adapt. Syst. 1 (2) (2006) 223–259.

[39] A. Filieri, C. Ghezzi, A. Leva, M. Maggio, Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements, in: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 283–292.

[40] F. Křikava, P. Collet, R. B. France, Actor-based runtime model of adaptable feedback control loops, in: Proceedings of the 7th Workshop on Models@Run.Time, MRT '12, ACM, New York, NY, USA, 2012, pp. 39–44.

[41] F. Krikava, P. Collet, R. B. France, ACTRESS: domain-specific modeling of self-adaptive software architectures, in: Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014, 2014, pp. 391–398.

[42] D. Garlan, S. Cheng, A. Huang, B. R. Schmerl, P. Steenkiste, Rainbow: Architecture-based self-adaptation with reusable infrastructure, IEEE Computer 37 (10) (2004) 46–54.

[43] S. Cheng, D. Garlan, Stitch: A language for architecture-based self-adaptation, Journal of Systems and Software 85 (12) (2012) 2860–2875.

[44] S. Cheng, D. Garlan, B. R. Schmerl, Architecture-based self-adaptation in the presence of multiple objectives, in: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, SEAMS 2006, Shanghai, China, May 21-22, 2006, 2006, pp. 2–8.

[45] J. Cámara, A. Lopes, D. Garlan, B. R. Schmerl, Adaptation impact and environment models for architecture-based self-adaptive systems, Sci. Comput. Program. 127 (2016) 50–75.

[46] D. Weyns, S. Malek, J. Andersson, On decentralized self-adaptation: Lessons from the trenches and challenges for the future, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '10, ACM, New York, NY, USA, 2010, pp. 84–93.

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

[48] N. Khakpour, S. Jalili, M. Sirjani, U. Goltz, B. Abolhasanzadeh, {HPobSAM} for modeling and analyzing {IT} ecosystems through a case study, Journal of Systems and Software 85 (12) (2012) 2770 – 2784, self-Adaptive Systems.

[49] M. Sirjani, A. Movaghar, A. Shali, F. S. de Boer, Modeling and verification of reactive systems using rebeca, Fundam. Inf. 63 (4) (2004) 385–410.

[50] P. Arcaini, E. Riccobene, P. Scandurra, Modeling and analyzing mape-k feedback loops for self-adaptation, in: Proceedings

of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 13–23.

[51] P. Arcaini, E. Riccobene, P. Scandurra, Formal design and verification of self-adaptive systems with decentralized control, ACM Trans. Auton. Adapt. Syst. 11 (4) (2017) 25:1–25:35.

[52] J. Cámara, G. A. Moreno, D. Garlan, Stochastic game analysis and latency awareness for proactive self-adaptation, in: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, ACM, New York, NY, USA, 2014, pp. 155–164.

[53] K. Angelopoulos, A. V. Papadopoulos, V. E. Silva Souza, J. Mylopoulos, Model predictive control for software systems with cobra, in: Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '16, ACM, New York, NY, USA, 2016, pp. 35–46.

[54] G. A. Moreno, A. V. Papadopoulos, K. Angelopoulos, J. Cámara, B. Schmerl, Comparing model-based predictive approaches to self-adaptation: Cobra and pla, in: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 42–53.